Lecture Notes in Computer Science 3167

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

Ana Cavalcanti
Augusto Sampaio
Jim Woodcock (Eds.)

Refinement Techniques in Software Engineering

First Pernambuco Summer School
on Software Engineering, PSSE 2004
Recife, Brazil, November 23-December 5, 2004
Revised Lectures

Authors

Ana Cavalcanti
University of York
Department of Computer Science
Heslington, York YO10 5DD, UK
E-mail: Ana.Cavalcanti@cs.york.ac.uk

Augusto Sampaio
Federal University of Pernambuco
Centre for Informatics
CEP 50740-540, Recife-PE, Brazil
E-mail: acas@cin.ufpe.br

Jim Woodcock
University of York
Department of Computer Science
Heslington, York YO10 5DD, UK
E-mail: jim@cs.york.ac.uk

Library of Congress Control Number: 2006933059

CR Subject Classification (1998): D.2, D.1, F.3, K.6.3

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-540-46253-8 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-46253-8 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11889229 06/3142 5 4 3 2 1 0

Preface

The *Pernambuco School on Software Engineering (PSSE) 2004* was the first in a series of events devoted to the study of advanced computer science and to the promotion of international scientific collaboration. The main theme in 2004 was *refinement* (or *reification*). Refinement describes the verifiable relationship between a specification and its implementation; it also describes the process of discovering appropriate implementations, given a specification. Thus, in one way or another, refinement is at the heart of the programming process, and so is the major daily activity of every professional software engineer. The Summer School and its proceedings were intended to give a detailed tutorial introduction to the scientific basis of this activity.

These proceedings record the contributions from the invited lecturers. Each chapter is the result of a thorough revision of the initial notes provided to the participants of the school. The revision was inspired by the synergy generated by the opportunity for the lecturers to present and discuss their work among themselves, and with the school's attendees. The editors have tried to produce a coherent view of the topic by harmonizing these contributions, smoothing out differences in notation and approach, and providing links between the lectures. We apologize to the authors for any errors introduced by our extensive editing.

Although the chapters are linked in several ways, each one is sufficiently self-contained to be read in isolation. Nevertheless, Chap. 1 should be read first by those interested in an introduction to refinement.

Chapter 1. We begin by setting the scene, introducing ideas and notations that are taken as general background by the lecturers. We discuss program semantics, using Dijkstra's language of guarded commands as an illustration. We start with program assertions—perhaps the most widely used formal method in programming (assertions form about 1% of Microsoft Windows code)—and then continue with predicate transformers. We describe the basic notions of refinement, and discuss a simple refinement algebra. Then we relate this to program development by formalizing the process of stepwise refinement of specifications into programs. Finally, we describe a useful mathematical structure: the lattice of specifications ordered by refinement. Each of the following four chapters uses these ideas to illustrate refinement for a given paradigm: object orientation, concurrency, probabilistic programs, and real-time and fault-tolerant systems.

Chapter 2. Sampaio and Borba describe refinement in the object-oriented setting, illustrating the ideas using sequential Java, although their work is of general applicability. Their approach is algebraic: they give laws for reasoning about and refining object-oriented programs. They discuss soundness with respect to predicate transformer semantics, and demonstrate completeness by showing that their set of laws is comprehensive enough to be able to reduce any program to a nor-

mal form. Finally, they show how their laws can be used to refactor programs, in order to adapt their structure while preserving their semantics.

Chapter 3. Davies describes refinement of concurrent and distributed systems using the notations of CSP. He starts by introducing CSP as a process algebra, with a set of operators and a rich collection of laws for reasoning about the behavior of processes. The denotational semantics of the language is used to give a simple notion of refinement between processes: every behavior of an implementation must be a specified behavior. These ideas are explored through an example involving protocols and their service specifications.

Chapter 4. McIver and Morgan add probabilistic nondeterminism to Dijkstra's guarded command language. They make a corresponding change to the programming logic, replacing weakest preconditions by greatest pre-expectations. These are generalizations of predicates that can be used to express the probability that a program achieves a postcondition. They explain how we can extend standard reasoning concepts like invariants and variants to handle probabilistic programs. They give a series of examples and two longer case studies.

Chapter 5. Liu and Joseph deal with refinement in real-time and fault-tolerant systems. They use transition systems as their computational model and Lamport's "Temporal Logic of Actions (TLA)" as a specification language in reasoning about functional correctness, timing properties, fault-tolerance, and schedulability. Their work is explained through an example of the interface between a processor and a memory device.

Chapter 6. Cavalcanti and Woodcock introduce "Unifying Theories of Programming" as a uniform foundation for all these paradigms. They give a tutorial introduction to an alphabetized version of Tarksi's relational calculus. They show how this leads to a simple denotational semantics of a language of terminating programs, and show that they form a complete lattice. They extend this work to Hoare-He designs, a relational model of pre- and postcondition specifications, exploring the space of designs as a subtheory of relations characterized by certain healthiness conditions. Then they turn their attention to another relational subtheory—reactive processes—once again characterized by healthiness conditions. Finally, they show that the reactive image of the design lattice gives a suitable semantic model for CSP. They end by comparing this semantics with the model given by Davies in his chapter. After this survey of refinement and its different theories, the final two chapters are on mechanical or automated support for refinement.

Chapter 7. Clayton and O'Halloran describe the practice of refinement in industry. They have designed the "Compliance Notation" for demonstrating the refinement relation between software and its specification. They have built a tool to support this demonstration, an essential item for industrial-scale application of refinement. They describe an extended example of a correctness argument for

programs written in the SPARK Ada subset. They present an application involving the correct implementation of control laws that govern control systems.

Chapter 8. Déharbe presents a very successful approach to verification, whose high level of automation has made it very attractive to industry. This chapter presents the main temporal logics used for specification of properties, and the main structures and algorithms used in tools. Widely used tools like SPIN and SMV are discussed. The approach is briefly compared with that adopted for model checking of CSP processes.

We are grateful to the members of the Organizing Committee, who worked very hard to provide an enjoyable experience for all of us. Without the support of our sponsors, *PSSE 2004* could not have been a reality. Their recognition of the importance of this event for the Software Engineering community in Latin America is greatly appreciated. We would also like to thank all the lecturers for their invaluable technical and scientific contribution, and for their commitment to the event; the effort of all authors is greatly appreciated. Finally, we are grateful to all the participants of the school. They are the main focus of the whole event.

April 2006

Ana Cavalcanti
Augusto Sampaio
Jim Woodcock

Organization

PSSE 2004 was organized by the Centro de Informática, Universidade Federal de Pernambuco (CIn/UFPE), Brazil, in cooperation with the United Nations University, International Institute for Software Technology (UNU/IIST), and the University of York, UK.

Executive Committee

Ana Cavalcanti	University of York
Antonio Cerone	UNU/IIST
Zhiming Liu	UNU/IIST
Augusto Sampaio	CIn/UFPE *(Managing Director)*
Jim Woodcock	University of York

Sponsoring Institutions

Formal Methods Europe
Sociedade Brasileira de Computação,Brazil
United Nations University, Macau
Universidade Federal de Pernambuco (CIn/UFPE), Brazil
University of York, UK

Acknowledgements

Paulo Borba and Augusto Sampaio. Several parts of their chapter were extracted and adapted from their previous joint work with Ana Cavalcanti and Márcio Cornélio: "Algebraic reasoning for object-oriented programming", *Science of Computer Programming*, 52:53–100, 2004. They thank their collaborator David Naumann for many discussions that significantly contributed to the work they report here, which was partially carried out when the authors were visiting the Stevens Institute of Technology at New Jersey, USA. They also thank Tiago Massoni and Rohit Gheyi for several important comments on an earlier version of their chapter, and Leila Silva for discussions concerning the impact of reference semantics in the proposed laws for their language. They are partially supported by the Brazilian Research Agency, CNPq, grants 521994/96–9 (Paulo Borba), 521039/95–9 (Augusto Sampaio), and 680032/99-1 (DARE CO-OP project, jointly funded by CNPq PROTEM-CC and the National Science Foundation).

Annabelle McIver and Carroll Morgan. Their chapter reports work carried out with Jeff Sanders, Thai Son Hoang and Karen Seidel. It was supported in the UK by the *EPSRC* and in Australia by the *ARC*. An earlier version of Sect. 1–5 of their chapter first appeared in the *South African Computer Journal* [42],who have graciously allowed it to be reprinted. Section 6 is new; it is based on (and extends) one of the lectures of the Summer School. The *SACJ* article was in turn a "transliteration" of a still earlier work [194] concerning probabilistic *Generalized Substitutions* [3]Chapters 1,2 and Sect. 3.1 of a text [181] by the same authors provides further insight into the material of Sect. 1–5.

Zhiming Liu and Mathai Joseph. Zhiming Liu's work has been supported by the UNU-IIST Research Project on Formal Methods of Object and Component Systems and the project HighQSoftD funded by Macao Science and Technology Fund, and the Chinese NSF project 60573085.

Ana Cavalcanti and Jim Woodcock. Their contribution is partially funded by the Royal Society of London and by QinetiQ Malvern, but their greatest debt is to Tony Hoare and He Jifeng for their inspirational work in unifying theories. The authors were the *Formal Methods Europe Lecturers* at the *PSS 2004*, and are grateful to FME for their support of this event, where their chapter was first presented. Earlier versions of the material were presented at the Universities of Oxford, Kent, and York, at QinetiQ Malvern, at the Danish Technical University, and as an invited tutorial at *Integrating Formal Methods, IFM 2004*. The authors have benefited from discussions with Chen Yifeng about closure; their proof of ***R***2-L0 is based on his original idea. Augusto Sampaio made a large number of detailed and useful comments on the technical material in the chapter.

Phil Clayton and Colin O'Halloran. Simulink is a registered trademark.

David Déharbe. His chapter subsumes and improves on his work in *Logic for Concurrency and Synchronisation*, volume 18 of Trends in Logic.

Table of Contents

Refinement: An Overview

Ana Cavalcanti[1], Augusto Sampaio[2], and Jim Woodcock[1]

[1] Department of Computer Science
University of York
York, UK
[2] Centro de Informática
Universitates Federal de Pernambuco
Recife - PE, Brazil

The purpose of this initial chapter is to introduce concepts and techniques assumed as general background in the remaining chapters of this book. The relevant notions are introduced using a simple and well-known programming notation: Dijkstra's language of guarded commands [81], presented in Section 1.

Three classical approaches to assigning semantic meaning to programs are then explored. In Section 2 we discuss the annotation of programs with assertions and the associated reasoning framework (Hoare Logic). Section 3 is dedicated to a calculational style where the behaviour of a program is defined in terms of a predicate transformer: its weakest precondition. Partial and total correctness of programs are contrasted in these two sections. The important notion of program refinement is introduced in Section 4. We start with some intuition and then we give a weakest precondition based definition, followed by an alternative (but equivalent) definition in terms of nondeterminism.

In Section 5, we explore another approach to program semantics, known as refinement algebra, which is based on equations and inequations (laws) relating programming constructs; algebraic laws allow a term rewriting style of program transformation. We then show, in Section 6, how the programming constructs can be embedded into a more abstract space of specifications; we introduce Morgan's specification statement and illustrate Morgan's refinement calculus concerning both algorithmic and data refinement. In Section 7 we discuss how a programming (or specification) language with a refinement ordering can be regarded as a lattice. This allows using well-established results of lattice theory in programming methodologies. We conclude this chapter with a brief discussion of refinement in other programming paradigms and the importance of tools to support program refinement in practice.

1 A Simple Programming Notation

The version of Dijkstra's Guarded Command Language (GCL) adopted here is summarised below; c stands for a command, x for a list of variables, e for a list of expressions, and ψ for a predicate.

A. Cavalcanti, A. Sampaio, and J. Woodcock (Eds.): PSSE 2004, LNCS 3167, pp. 1–17, 2006.

$c ::= \mathbf{skip} \mid \mathbf{abort}$	do nothing, abortion
$\mid x := e \mid c;\ c$	assignment, sequence
$\mid \mathbf{if}\ [] i \bullet \psi_i \rightarrow c_i\ \mathbf{fi}$	conditional
$\mid \mathbf{do}\ [] i \bullet \psi_i \rightarrow c_i\ \mathbf{od}$	iteration
$\mid c \sqcap c$	nondeterminism
$\mid \mathbf{var}\ x : T \bullet c$	local variable block

The primitive constructs are standard. The command **skip** has no effect and, when executed, terminates immediately. In contrast, the command **abort** has a completely arbitrary behaviour; rather than being deliberately written by a programmer, it may arise as a result of some undesirable computation like division by zero, or an infinite loop without any externally visible effect.

The remaining primitive command of GCL is assignment. The program $x := e$ is a multiple (or simultaneous) assignment, where x is a list of distinct variables and e an equal-length list of expressions. The components of e are evaluated and assigned to the corresponding components of x in the same position. All the contributing chapters of this book assume that expressions have no side-effect. In this chapter we further assume that expressions are always well-defined (their evaluation is always successful). In Chapter 2, on refinement of object-oriented programs, we consider assignments whose expressions might fail when evaluated, and the impact of side effects in expressions is discussed. As a simple example of a multiple assignment, the command

$$x, y := y, x$$

swaps the values of x and y.

In GCL, the body of the conditional is a guarded command set. A guarded command takes the form $\psi \rightarrow c$, where ψ, the guard, is a predicate. The choice of which command executes is between those whose guards evaluate to true. If more than one guard is satisfied, the choice is nondeterministic; if no guard evaluates to true, the conditional behaves like **abort**. As an example, we consider the following command that assigns to z the greatest value held by x or y.

$$\begin{array}{l}\mathbf{if}\ (x \leq y) \rightarrow z := y \\ []\ (y \leq x) \rightarrow z := x \\ \mathbf{fi}\end{array}$$

When $x = y$, the choice of which assignment executes is nondeterministic.

The body of an iteration ($\mathbf{do}\ [] i \bullet \psi_i \rightarrow c_i\ \mathbf{od}$) is also a guarded command set. Similarly to the conditional, the choice of which guarded command executes depends on the evaluation of the guards. If more than one guard is satisfied, the choice is nondeterministic; if no guard evaluates to true, the iteration terminates successfully, behaving like **skip**. In the program fragment below, the final value of r is the factorial of the natural number assigned to n.

$$r := 1;\ \mathbf{do}\ (n > 1) \rightarrow r := r * n;\ n := n - 1\ \mathbf{od}$$

The program $c_1 \sqcap c_2$ denotes an arbitrary (also known as *demonic*) choice between the commands c_1 and c_2, in the sense that either one can be selected for execution. For instance, consider the program fragment below.

$x := 1 \sqcap x := 2$;
if $(x = 1) \rightarrow$ **skip**
$[]$ $(x = 2) \rightarrow$ **abort**
fi

In this context, either $x := 1$ or $x := 2$ will be selected, so that is possible that the execution of the conditional leads to abortion. It is important to observe, however, that, according to equality and refinement notions that reflect total correctness, any program that might abort (like the one above) is actually identified with abort.

Some specification languages (like the one introduced in the next chapter) include a complementary notion of nondeterminism known as *angelic* choice. In this case the most suitable command for a given context is selected for execution. If the choice in the above example were angelic, the assignment $x := 1$ would have been selected. Operationally, the view is that of *backtracking* in the search for the best possible execution of the program. For further considerations on demonic and angelic choices see, for instance, [17].

The program (**var** $x : T \bullet c$) declares the variable x of type T for use in the command c. Local blocks of this form may appear anywhere a command is expected. The occurrence of a variable x in the scope of a local declaration is *bound*, and *free* otherwise. For example, y is bound in **var** $y : T \bullet x := y$, but free in $x := y$. The program fragment that computes the factorial of n, previously presented, can be redesigned to leave n untouched, using a local variable.

$$r := 1;\ \textbf{var}\ t : \mathbb{N} \bullet t := n;\ \textbf{do}\ (n > 1) \rightarrow r := r * t;\ t := t - 1\ \textbf{od}$$

In the following two sections we discuss two well-established approaches to define a formal semantics to programming languages; the guarded command language introduced in this section is used as illustration.

2 Assertions and Hoare Logic

A classical approach to assigning formal meaning to (and reasoning about) programs, well-known as Hoare logic, is presented in [114]. Like in other branches of mathematics, the basis of this approach is to define the behaviour of programming constructs in terms of axioms and inference rules. Axioms define the semantics of the primitive commands like **skip**, **abort** and assignment. Each axiom takes the form of a Hoare triple, $P\,\{c\}\,Q$, where c is a command and P and Q are logical assertions, playing the role of pre- and postcondition, respectively. A Hoare triple is interpreted as follows: if P is true and c terminates successfully, then Q must be established. The semantics of language operators like sequential composition and conditional is defined by inference rules which assume that a Hoare triples hold for the arguments.

The original formulation of Hoare logic is for *partial correctness*, which means that the axioms and inference rules assume successful termination of a program, as can be inferred from the above interpretation. Nevertheless, several subsequent formulations of Hoare logic as, for instance, [74, 106, 203], address *total correctness*. In this case, the interpretation of a triple $P\,\{c\}\,Q$ is: if P is true, then c must terminate successfully and establish Q. While the semantics described here regards partial correctness, the weakest precondition semantics defined in the next section embodies termination.

The Hoare triples for the language presented in the previous section are summarised in Table 1.

Table 1. Hoare triples for GCL

$P\,\{\mathbf{skip}\}\,P$
$\mathbf{true}\,\{\mathbf{abort}\}\,\mathbf{false}$
$P[e/x]\,\{x := e\}\,P$
If $P\,\{c_1\}\,Q$ and $Q\,\{c_2\}\,R$ then $P\,\{c_1;\ c_2\}\,R$
If, for all i, $(\psi_i \wedge P)\,\{c_i\}\,Q$ then $P\,\{(\mathbf{if}\ [\!]\, i \bullet \psi_i \rightarrow c_i\ \mathbf{fi})\}\,Q$
If, for all i, $(\psi_i \wedge P)\,\{c_i\}\,P$ then $P\,\{\mathbf{do}\ [\!]\, i \bullet \psi_i \rightarrow c_i\ \mathbf{od}\}\,(P \wedge (\bigwedge i \bullet \neg\psi_i))$
If $P\,\{c_1\}\,Q$ and $P\,\{c_2\}\,Q$ then $P\,\{c_1 \sqcap c_2\}\,Q$
If $P\,\{c\}\,Q$ then $(\forall\, x : T \bullet P)\,\{\mathbf{var}\ x : T \bullet c\}\,(\exists\, x : T \bullet Q)$

As **skip** has no effect, any logical assertion that is true before its execution, remains true after it terminates. Being totally unpredictable, nothing can be guaranteed concerning what **abort** could establish as postcondition, if it terminates. The axiom for assignment formalises that if P is to be established after the assignment of e to x, then the assertion obtained from P, by replacing all free occurrences of x with e, must be true before the assignment.

The semantics of the remaining constructs is defined by inference rules. For $c_1;\ c_2$, the precondition is that of c_1, and the postcondition is that established by c_2; furthermore, the postcondition established by c_1 coincides with the precondition of c_2. Concerning the conditional, each of its guarded commands must establish the expected postcondition, provided the corresponding guard is true. In the case of a nondeterministic execution of c_1 and c_2, the expected result can only be ensured if both produce such a result.

The semantics of iteration is based on an invariant P that is assumed to hold for each guarded command in the body of the loop, whenever the corresponding guard is true. This invariant is also assumed to hold before the entire iteration starts executing. Then, P must also hold after (possible) termination of the iteration, when none of the guards holds any longer.

Assuming that an assertion holds for the body of a local declaration block (where occurrences of a variable, say x, might be free), the precondition for the entire block is assumed to hold for all possible values of x. Then there must be at least one possible value of x such that the postcondition holds. When P and Q do not mention x the quantifiers have no effect and can be eliminated.

Apart from the axioms and inference rules that define the semantics of the programming constructs, Hoare logic includes additional rules for reasoning, like the *rules of consequence* [114] displayed below.

If $Q\,\{c\}\,R$ and $P \Rightarrow Q$ then $P\,\{c\}\,R$
If $Q\,\{c\}\,R$ and $R \Rightarrow S$ then $P\,\{c\}\,S$

As an example of the application of these rules, we can observe that the behaviour of **abort** is really unpredictable, since a Hoare triple such as

$$x = 1\ \{\mathbf{abort}\}\ x = 2$$

holds. The proof follows from the fact that $x = 1 \Rightarrow \mathbf{true}$ and $\mathbf{false} \Rightarrow x = 2$.

3 Weakest Preconditions

The seminal work [81] presented an alternative technique to reason about programs: weakest preconditions calculus. In this approach, the semantics of a program is characterised by a predicate transformer: a function from predicates to predicates usually called *wp*. When applied to a program p and to predicate ψ, *wp* gives a predicate that defines all states in which execution of p terminates and leads to a state in which ψ holds. The predicate ψ is called a postcondition, and $wp.p.\psi$ is the weakest precondition that guarantees that the program establishes ψ; a period is used here to denote function application. In contrast with the previous chapter, here we consider total correctness.

The weakest precondition semantics of the simple language presented in Section 1 is shown in Table 2. The semantics of a loop is given in terms of the semantics of recursion, which is discussed in Section 7.

Table 2. Weakest precondition semantics of GCL

$wp.\mathbf{skip}.\psi$	ψ
$wp.\mathbf{abort}.\psi$	$false$
$wp.x := e.\psi$	$\psi[e/x]$
$wp.(c_1;\ c_2).\psi$	$wp.c_1.(wp.c_2.\psi)$
$wp.(\mathbf{if}\ []i \bullet \psi_i \rightarrow c_i\ \mathbf{fi}).\psi$	$(\bigvee i \bullet \psi\_i) \wedge (\bigwedge i \bullet \psi_i \Rightarrow wp.c_i.\psi)$
$wp.(c_1 \sqcap c_2).\psi$	$(wp.c_1.\psi) \wedge (wp.c_2.\psi)$
$wp.(\mathbf{var}\ x : T \bullet c).\psi$	$\forall x : T \bullet wp.c.\psi$

Since **skip** terminates, but does not affect any variables, the only way in which it can establish a postcondition ψ is if it already holds. Because **abort** may not even terminate, it can never provide a guarantee to establish any postcondition. The assignment $x := e$ establishes a postcondition ψ if it holds when the variables x take the values e (and all other variables are not changed).

The semantics of sequence is function composition. The weakest precondition for $c_1 ; c_2$ to establish ψ is the weakest precondition for c_1 to establish the weakest precondition for c_2 to establish ψ.

For a conditional to be guaranteed at least to terminate, one of its guards has to be true. Moreover, if it is to be guaranteed that it establishes ψ, then the weakest precondition for each of the commands c_i associated with guards ψ_i that are true have to be satisfied. This is because any of these commands may be chosen for execution.

This also explains the semantics of the choice $c_1 \sqcap c_2$. If it is to be guaranteed that it establishes ψ, then both c_1 and c_2 have to provide the guarantee. Finally, arbitrary choice is also embedded in the semantics of a variable block ($\mathbf{var}\ x : T \bullet c$); the initial value of x is nondeterministically chosen. It is only guaranteed to establish ψ, if c does, for every value that x may take.

4 Refinement Notions

During development, sometimes the resulting program does not behave exactly as the original program, but is possibly better, from the point of view of the user. In this case, we say that we have a refinement of the original program. Formally, an ordering relation on programs is used: $c_1 \sqsubseteq c_2$ holds when c_2 is at least as good as c_1 in the sense that it will meet every purpose and satisfy every specification satisfied by c_1.

A refinement relation is a pre-order: it is reflexive and transitive.

Law 1 (Refinement reflexive). $c \sqsubseteq c$

Law 2 (Refinement transitive). $(c_1 \sqsubseteq c_2) \land (c_2 \sqsubseteq c_3) \Rightarrow (c_1 \sqsubseteq c_3)$

Often, and in this book, $\sqsubseteq$ is a partial ordering, further satisfying the antisymmetry law.

Law 3 (Refinement antisymmetric). $(c_1 \sqsubseteq c_2) \land (c_2 \sqsubseteq c_1) \Rightarrow (c_1 = c_2)$

While the transitivity property of the refinement relation supports stepwise refinement, antisymmetry reduces proofs of equivalence to proofs of mutual refinement, just like equivalence of predicates in the predicate calculus can be established using mutual implication.

Apart from these properties, to allow compositional transformations (independent refinement of subcomponents of compound programs) the language operators should preferably be monotonic with respect to $\sqsubseteq$. For example, $c_1 \sqsubseteq c_2$ must imply $c_1 ; c_3 \sqsubseteq c_2 ; c_3$. In general, we have the result below, where F is a context: a function on programs built from the language operators.

Law 4 (Refinement compositional). $(c_1 \sqsubseteq c_2) \Rightarrow (F(c_1) \sqsubseteq F(c_2))$

Ideally, this must hold for all valid contexts F. Some languages, nevertheless, allow constructs which are not monotonic with respect to $\sqsubseteq$, and therefore re-

strictions must be imposed on F so that the above law holds. This is the case, for instance, of private attributes in object-oriented languages. They cannot be used to build contexts since an improved class does not necessarily has the same private attributes of the original class.

A formal definition of the refinement relation can be given in the weakest precondition model, in a simple and intuitive way. A refined program must work in at least the same set of states as the original program, but possibly in a larger set. The stronger a predicate is, the smaller is the set of elements it defines. Therefore, for all postconditions ψ, the weakest precondition of a refined program must be no stronger than that of the original program.

$$(c_1 \sqsubseteq c_2) \mathrel{\widehat{=}} wp.c_1.\psi \Rightarrow wp.c_2.\psi$$

Refinement can also be understood as a reduction of nondeterminism. Therefore, if the nondeterministic choice between c_1 and c_2 always yields c_1, this means that c_2 refines c_1. This gives an alternative characterisation of $\sqsubseteq$.

$$(c_1 \sqsubseteq c_2) \mathrel{\widehat{=}} (c_1 \sqcap c_2 = c_1)$$

Nondeterminism is used in specifications to provide abstraction: choices that are better made during design or implementation are left open.

Exercise 1. Derive the above definition of refinement from the previous one and the weakest precondition semantics of $\sqcap$ given in Section 2.

5 Refinement Algebra

Program transformation with the preservation of semantics can be formally justified in terms of a semantic model like weakest precondition or Hoare logic, as discussed in previous sections. For instance, a program c_1 can be safely transformed into a program c_2 provided c_1 and c_2 have the same weakest precondition; the transformation is also valid if c_2 is a refinement of c_1 (the weakest precondition of c_1 implies that of c_2).

In an algebraic style of reasoning, the properties of the programming constructs are captured by equations and inequations (laws) that directly relate these constructs. An attractiveness of algebraic reasoning, therefore, is that it is entirely conducted at the programming level; at least in principle, this seems more appealing for programmers. In this approach, given that the algebraic laws are sound, transformations based on their application are also correct by construction. Soundness of the laws is considered a separate issue; this is done by proving the laws in a mathematical model, like weakest precondition. The focus here is on the presentation of the laws, rather than on their proofs.

First we consider simple properties of **skip**, assignment and sequential composition. For example, the following law states that the assignment of the value of a variable to itself has no effect.

Law 5 (Void assignment). $(x := x) = \mathbf{skip}$

Such a vacuous assignment can also occur as part of a multiple assignment.

Law 6 (Identity assignment). $(x, y := e, y) = (x := e)$

The list of variables and expressions may be subjected to the same permutation, without changing the effect of the assignment.

Law 7 (Assignment symmetry). $(x, y := e, f) = (y, x := f, e)$

Two assignments to the same variables can be readily combined into a single assignment.

Law 8 (Combine assignments). $(x := e;\ x := f) = (x := f[e/x])$

The notation $f[e/x]$ denotes the substitution of e the free occurrences of x in f.

As **skip** has no effect, it is both the left and the right identity of sequence.

Law 9 (Composition identity). $(\mathbf{skip};\ c) = c = (c;\ \mathbf{skip})$

A comprehensive set of laws for imperative programming can be found in [116]. The purpose here is to illustrate the algebraic reasoning style. As a simple example, we prove that assignments can be swapped when there is no interference.

Example 1 (Swap assignments). Consider x, y, w and z are distinct identifiers. Then $(x := y;\ w := z) = (w := z;\ x := y)$

Proof.

$$
\begin{array}{ll}
x := y;\ w := z & \text{[Law 6]} \\
x, w := y, w;\ w, x := z, x & \text{[Law 7]} \\
w, x := w, y;\ w, x := z, x & \text{[Law 8]} \\
w, x := z, y & \text{[Law 8]} \\
w, x := z, x;\ w, x := w, y & \text{[Law 7]} \\
w, x := z, x;\ x, w := y, w & \text{[Law 6]} \\
w := z;\ x := y &
\end{array}
$$

The laws allow us to prove that the two sequences of assignments (although syntactically different) behave the same and, therefore, are semantically equivalent.

A nice feature of the algebraic style is modularity. One can explore program properties incrementally, considering one construct at a time. For example, let us now deal with variable declaration. A simple property is that, if the declared variable is not used in its scope, then the declaration has no effect.

Law 10 (Void declaration). $(\mathbf{var}\ x : T \bullet c\ ) = c$ **provided** *x is not free in c*

Recall that an occurrence of a variable x in c is *bound* (or local) if it is in the scope of a declaration of x in c, and *free* (or global) otherwise.

Another property of local variable declaration is that assigning to a variable at the end of its scope has no effect.

Law 11 (Assignment elimination).

$$(\mathbf{var}\ x : T \bullet c;\ x, y := e, f\ ) = (\mathbf{var}\ x : T \bullet c;\ y := f\ )$$

As is usual in an algebraic presentation, the introduction of the new laws for declaration has no impact on the previous laws; actually they contribute to the set of properties that hold of our simple programming language. Therefore, the proof of Example 1 does not need to be revised. Transformations involving declarations can now be performed using the previous and the new laws. The following exercise serves as an illustration.

Exercise 2. Assuming that z is not free in $x := y$, and that these variables have type T, prove the following equivalence:

$$(\mathbf{var}\ z : T \bullet z := y;\ x := z\ ) = (x := y)$$

As previously discussed, during program transformation, sometimes the resulting program does not behave exactly as the original program, but is possibly better than (a refinement of) it. The following is an example of a refinement law.

Law 12 (Declaration initialisation).

$$(\mathbf{var}\ x : T \bullet c\ ) \sqsubseteq (\mathbf{var}\ x : T \bullet x := e;\ c\ )$$

Since the initial value of a declared variable is totally arbitrary, initialisation of a variable may reduce nondeterminism, leading to a more predictable program.

Nondeterminism can be understood as allowing choices to be made. Program development usually starts with abstract specifications which leave several design decisions for the programmer to take. One important issue in refining a specification into a program is reducing nondeterminism. This is addressed in further detail in the next section.

6 Specification and Program Development

While the notation introduced so far exemplifies well-known (executable) programming constructs, it is not suitable for writing abstract specifications. In the view followed by consolidated approaches to program development, a mathematical trick is applied: the programming language is embedded within a more general specification notation. In this way, a single notation is used both for programming and for specification; programs appear as a special kind of specification. Therefore, program development reduces to transformations of specifications within a uniform framework. Examples of approaches which adopt this view are the refinement calculi by Back [14], Morgan [192] and Morris [199].

A distinguishing feature of Morgan's calculus is the *specification statement*:

$$w : [pre, post]$$

which describes a program that, when executed in a state satisfying the precondition *pre*, terminates in a state satisfying the postcondition *post*, possibly modifying the values of variables in the list (frame) w.

As an example, consider the specification statement

$$s, r : [e \notin s, s = s_0 \cup \{e\} \land r = \texttt{"}Okay\texttt{"}]$$

whose effect is to add a new element e to a set s, and assign to r the constant "*Okay*", indicating successful execution of the operation. By convention, occurrence of framed variables in the precondition refer to their initial values, whereas in the postcondition such occurrences refer to the final values of the framed variables. To reference initial values of framed variables in the postcondition, a subscript is adopted. Therefore, s_0 stands for the initial value of s in the postcondition above.

This specification can be refined into executable code, as discussed in the sequel. In this way, the language allows us to start with an abstract specification of a program and progressively refine it by mixing code and specifications, and then finally obtain a program with executable constructs only.

Some extreme specifications are of particular interest for reasoning. For example, we can write **abort** as a specification.

$$\mathbf{abort} = x : [\mathbf{false}, \mathbf{true}]$$

It is the worst possible specification. It is never guaranteed to terminate (precondition **false**), and even when it does, its outcome is completely arbitrary (postcondition **true**). It allows any refinement; for instance, programs setting x to arbitrary values. At the other extreme, we have the best possible specification

$$\mathbf{miracle} = x : [\mathbf{true}, \mathbf{false}]$$

which can execute in any state (precondition **true**) and establishes as outcome the impossible postcondition **false**. This is an infeasible specification; it cannot be realised as an executable program. In fact, it is not refined by any other specification or code. So, arriving at this specification during development indicates that the developer should return to a previous development step and make alternative design choices in order to be able to implement the initial specification.

It is also useful in program derivation or transformation to assume that a condition b holds at a given point in the program text. This can be written as $\{b\}$, and defined as follows.

$$\{b\} \mathrel{\widehat{=}} : [b, \mathbf{true}]$$

If b is **false**, $\{b\}$ reduces to **abort**. Otherwise, it behaves like **skip**: always terminates and does nothing. In [192], and in this book, this is called an *assumption*; it coincides with the concept of *assertion* in the setting of Hoare logic.

We can also give a simple specification to **skip**.

$$\mathbf{skip} =: [\mathbf{true}, \mathbf{true}]$$

The empty frame guarantees that no variables are changed.

6.1 Algorithmic Refinement

Morgan's calculus is perhaps the most appealing to practising programmers, since it includes several laws that allow transforming specification statements into executable programs. Some laws relate specification statements. Two of these capture the notion of refinement in program development. The first states that a program can be made more applicable (defined for a larger domain or set of states) when refined; in other words, its precondition can be weakened.

Law 13 (Precondition weakening). $w : [pre, post] \sqsubseteq w : [pre', post]$
provided $pre \Rightarrow pre'$

Concerning the effect (postcondition), refinement might lead to a more deterministic or predictable program, as already discussed in the previous section.

Law 14 (Postcondition strengthening). $w : [pre, post] \sqsubseteq w : [pre, post']$
provided $pre[w_0/w] \wedge post' \Rightarrow post$

This states that strengthening the postcondition, in states which satisfy the precondition, leads to refinement. The substitution of w_0 for w in the proviso is necessary due to the convention that initial values of framed variables in the postcondition are subscripted.

A refinement of our example specification is to increase its applicability, recording an error message in r when the element is already in the set.

$$\begin{aligned} s, r : [\mathbf{true}, &(e \notin s_0 \wedge s = s_0 \cup \{e\} \wedge r = \text{"}Okay\text{"}]) \vee \\ &(e \in s_0 \wedge s = s_0 \wedge r = \text{"}AlreadyMember\text{"})] \end{aligned}$$

Whenever the precondition of the original specification is satisfied (the element is not in the set), the refined version exhibits exactly the same behaviour. When the element is already in the set, the values of z and r are not defined in the original specification, and therefore totally arbitrary. In the refined version, when this happens, s is not modified and r takes the defined value "*AlreadyMember*".

Exercise 3. Prove the refinement:

$$\begin{aligned} & s, r : [e \notin s, s = s_0 \cup \{e\} \wedge r = \text{"}Okay\text{"}] \\ \sqsubseteq \\ & s, r : [\mathbf{true}, (e \notin s_0 \wedge s = s_0 \cup \{e\} \wedge r = \text{"}Okay\text{"}]) \vee \\ & \qquad\qquad (e \in s_0 \wedge s = s_0 \wedge r = \text{"}AlreadyMember\text{"})] \end{aligned}$$

Most of the laws of Morgan's refinement calculus relate particular forms of specification statements to programming constructs, serving as tools to refine abstract

specifications into code which can be effectively executed. The process is known as algorithmic or control refinement. For example, the law below allows the introduction of an assignment.

Law 15 (Assignment introduction). $w, v : [pre, post] \sqsubseteq v := e$
provided $(v = v_0) \land pre \Rightarrow post[e/v]$

It states that if the value of the assigned expression is suitable to establish the postcondition, in contexts where the precondition holds, then the assignment is a valid implementation of such a specification. In the proviso, the condition $(v = v_0)$ is necessary due to the convention that initial values of framed variables in the postcondition are subscripted. For example, in a specification statement of the form $v : [\mathbf{true}, v = v_0 + 1]$, v_0 denotes the initial value of v in the postcondition, whereas in an assignment of the form $v := v + 1$ no subscript variables are used. Identifying $v = v_0$ in the formulation allows to justify this kind of refinement. For this example, the proof obligation is $v = v_0 \land \mathbf{true} \Rightarrow v + 1 = v_0 + 1$, which clearly holds.

As another example, consider the following law.

Law 16 (Following assignment).

$$w, v : [pre, post] \sqsubseteq w, v : [pre, post[e/v]];\ v := e$$

It allows the extraction of an assignment from a specification statement , but still keeps the remaining behaviour as a (modified) specification statement.

Exercise 4. Prove the following refinement:

$$s, r : [e \notin s, s = s_0 \cup \{e\} \land r = \texttt{"}Okay\texttt{"}] \sqsubseteq (s := s \cup \{e\};\ r := \texttt{"}Okay\texttt{"})$$

6.2 Data Refinement

Complementarily to algorithmic refinement, program development normally involves change of data representation. A typical development starts with a specification whose data structures are abstract and, possibly, not even available in the target programming language. As the development progresses, the abstract data types give rise to more concrete representations.

As a simple example, consider a specification statement similar to that of the previous section, which adds a new element to a set.

$$s : [e \notin s, s = s_0 \cup \{e\}]$$

Then consider a possible implementation using a sequence, as a concrete representation of a set.

$$t : [e \notin \mathsf{set}\ t, t = t_0 \frown \langle e \rangle]$$

We use $\mathsf{set}\ t$ to denote the set with the elements of the sequence t; $\langle e \rangle$ stands for the singleton sequence with element e, and $\frown$ represents sequence concatenation.

Intuitively, the latter specification refines the former, since sequences (lists or arrays) are well-known structures used for implementing sets. Furthermore, both operations have the same observable effect of adding a new element to the relevant data structure. Nevertheless, attempting to prove this refinement using the laws for weakening precondition and strengthening postcondition soon reveals that a direct comparison between the two statements is not possible at all. The reason is that they operate on different data spaces: a set s and a sequence t.

The missing connection is a relation between the abstract and the concrete states. For the particular example,

$$s = \mathsf{set}\ t$$

Based on this relation, it is possible to prove this data refinement. In general, relations between abstract and concrete states can be arbitrary. In many cases of practical interest, nevertheless, these relations are functional, and are called *abstraction functions*. In our example, the relation is functional and the abstraction function is set.

In Morgan's calculus, data refinement is formulated at the level of programming modules. A module includes state variables, a state initialisation, and procedures which act on the module state. Broadly, the technique involves adding the concrete variables to the module being data refined, making the abstract variables auxiliary, and then removing the auxiliary (abstract) variables. When the relation between abstract and concrete states is functional, these steps are combined into a single step.

In this view, our abstract module would include the declaration of the variable s (say, a set of natural numbers), a state initialisation (say, the empty set), and several operations, including the one to insert new elements into the set, as presented above. To proceed with the data refinement, several transformations are proposed by Morgan to deal with initialisation, assignments, specification statements, and so on.

To illustrate the technique, we present a transformation for specification statements. A specification statement

$$w, x : [pre, post]$$

becomes

$$w, z : [pre[\mathsf{af}\ z/x], post[\mathsf{af}\ z_0, \mathsf{af}\ z/x_0, x]]$$

where af stands for the relevant abstraction function, x for the abstract variables, and z for the concrete variables. Observe that this transformation replaces x with z in the frame, and occurrences of x in the pre and in the postcondition with $\mathsf{af}\ z$. In the postcondition, both the initial and final values of x need to be replaced.

Exercise 5. Formalise the data transformation of the statement previously presented: $s : [e \notin s, s = s_0 \cup \{e\}]$ into $t : [e \notin \mathsf{set}\ t, t = t_0 \frown \langle e \rangle]$ considering the abstraction function: $s = \mathsf{set}\ t$.

7 Refinement and Lattices

We have not given a semantics for recursion yet, and presented only a partial semantics for iteration in Section 2. In this section, we come to this point as part of a discussion of refinement as a partial order in a lattice of monotonic predicate transformers [199, 97, 16]. A partial order is a reflexive, anti-symmetric and transitive relation between elements of a set. If, given any two elements of the set, they are always related by the order in some way, then we have a total order. Refinement, however, is a partial order between programs.

To give the semantics and reason about simple (non-recursive) procedures, we can use a copy rule: basically, it allows calls to the procedures to be replaced with their bodies. As an example, we consider the program below, which defines a procedure *Inc*, and calls it twice.

$$\mathbf{proc}\ Inc \mathrel{\widehat{=}} x := x + 1 \bullet Inc;\ Inc$$

It is equivalent to $x := x + 1;\ x := x + 1$. In the case of a recursive procedure, however, this approach does not work. As a second example, we take a procedure *Sum* that adds the value of x to that of another variable y (and sets x to 0).

$$\begin{array}{l}\mathbf{proc}\ Sum \mathrel{\widehat{=}} \mathbf{if}\ (x = 0) \rightarrow \mathbf{skip} \\ \qquad\qquad [\!]\ (x > 0) \rightarrow y := y + 1;\ x := x - 1;\ Sum \\ \qquad\qquad \mathbf{fi} \\ \bullet\ y := 5;\ x := 10;\ Sum\end{array}$$

In the main program, we call *Sum*; if we replace this call by the body of *Sum*, another call of *Sum* is introduced, so the copy rule does not really sort out the reasoning problem.

The declaration of *Inc* can be seen as a definition of this procedure through the equation

$$Inc = x := x + 1;\ x := x + 1$$

In the case of *Sum*, this equation is

$$Sum = \mathbf{if}\ (x = 0) \rightarrow \mathbf{skip}\ [\!]\ (x > 0) \rightarrow y := y + 1;\ x := x - 1;\ Sum\ \mathbf{fi}$$

The body of *Sum* can be regarded as a context: a function from programs to programs, which can be defined as follows using the λ notation.

$$\lambda\, X \bullet \mathbf{if}\ (x = 0) \rightarrow \mathbf{skip}\ [\!]\ (x > 0) \rightarrow y := y + 1;\ x := x - 1;\ X\ \mathbf{fi}$$

Therefore, the equation that defines *Sum* requires that it is a program that, when given as argument to the above function, the result is itself. For any function F, the arguments X for which $F(X) = X$ are called fixed points of F. So, the equation for *Sum* requires it to be a fixed point of the function on programs characterised by its body.

The problem is that fixed points may not exist, or there may be lots of them. So, for the equation characterised by the declaration of a recursive procedure to

be a valid definition, we need to guarantee that there is at least one fixed point, and that, when there are several, we have a way of choosing one. With this end, it is usual taking lattices with various properties as semantic models.

A lattice is a set S with a partial order $\leq$ that satisfies a few extra properties. They are based on the existence of lower and upper bounds for certain subsets of S. For any subset T of S, an upper bound u of T is an element of S such that $t \leq u$ for every t in T. Similarly, a lower bound l is such that $l \leq t$, for every $t \in T$. The least upper-bound is an upper bound that is smaller than all others; likewise, the greatest lower-bound is a lower bound that is bigger than all others.

Definition 1 (Lattice and complete lattice). *A* lattice *is a partially ordered set, in which all* non-empty finite *subsets have both a least upper-bound (join) and a greatest lower bound (meet). A* complete lattice *is a lattice in which* all *subsets have both a join and a meet.* □

Every complete lattice has a bottom (smallest element) and a top (biggest element).

Example 2 (Complete lattices). Complete lattices are often found in mathematics and computer science. We give two examples drawn from mathematics.

1. The power set of a given set S ordered by inclusion forms a complete lattice. The least element is the empty set and the greatest element is S itself. Join is union and meet is intersection of subsets.
2. The set of natural numbers ordered by divisibility forms a complete lattice. Divisibility gives $m \sqsubseteq n$ exactly when $(\exists k \bullet k \times m = n)$, and so forms a partial order. The bottom of this lattice is the number *1*, since it exactly divides every other number. The top element of the lattice is *0*, since it can be divided exactly by every other number. The join of finite sets is given by the least common multiple; for infinite sets, the join will always be *0*. The meet is the greatest common divisor, and for infinite sets this may well be greater than *1*. For example, the set of all even numbers has *2* as the greatest common divisor. □

An example of a complete lattice drawn from the area of refinement is the set of monotonic predicate transformers ordered by refinement. A discussed in Section 3, we can use a function *wp* to characterise (the semantics of) programs. For a program p, $wp.p$ is a predicate transformer; the set of all such predicate transformers pt is a convenient model for programs. The partial order $\sqsubseteq$ for predicate transformers defined as

$$pt_1 \sqsubseteq pt_2 \mathrel{\widehat{=}} pt_1.\psi \Rightarrow pt_2.\psi, \text{ for all predicates } \psi$$

corresponds to the refinement relation defined in Section 4.

A function f from a set S with a partial order $\leq_S$ to a set T with a partial order $\leq_T$ is monotonic if, for every x and y in S, if $x \leq_S y$, then $f(x) \leq_T f(y)$.

For every program p, the function $wp.p$ is monotonic. The partial order considered for the set of predicates is implication. If a postcondition ψ_1 implies another postcondition ψ_2, we say that ψ_1 is stronger than ψ_2, because less states satisfy ψ_1, and every state that satisfies ψ_1 also satisfies ψ_2. In this case, for every program p, $wp.p.\psi_1$ implies $wp.p.\psi_2$; this is because if from a particular initial state p is guaranteed to establish ψ_1, then it is also guaranteed to establish ψ_2.

The bottom of the complete lattice of monotonic predicate transformers is **abort**; the top is **miracle**. The join operator the demonic choice, and the meet operator is angelic choice.

Well-known results establish the existence of fixed points for functions on complete lattices. For example, we know that every monotonic function on a complete lattice has a fixed point. Since the body of a recursive procedure is a monotonic function on programs (see Law 4), this is in the direction of what we need to give meaning to recursive procedures. What we still need to sort out is the fact that such functions may have several fixed points. An extreme example is an infinite recursion: $\mathbf{proc}\, Inf \mathrel{\widehat{=}} Inf \bullet \ldots$. The function defined by its body is the identity $\lambda X \bullet X$. Every program is a fixed point of this function.

In such situations, we take the least fixed point to be the definition of the recursive procedure; it is denoted by $\mu X \bullet F(X)$, where F is the body of the procedure written as a function of X. This is the least refined program that satisfies the equation characterised by the definition of the recursive procedure. From the point of view of program development, this is the natural solution, since, as already explained, we want to impose as few restrictions as possible on a specification, and leave design decisions open. The meaning of Inf, for example, is taken to be the least refined program: **abort**.

The Knaster-Tarski fixed point theorem is very much used since it gives an explicit characterisation for the least fixed point of a monotonic function on a complete lattice; it is stated below.

Theorem 1 (Knaster-Tarski fixed point). *For every monotonic function $F(X)$ on a complete lattice with order $\leq$*

$$\mu X \bullet F(X) = \sqcap \{ X : L \mid F(X) \leq X \}$$

We use $\sqcap S$ to denote the greatest lower-bound of the set S.

The greatest fixed point of F also exists; it is denoted by $\nu X \bullet X$, and it is possible to prove that $\nu X \bullet F(X) = \bigsqcup \{ X : L \mid X \leq F(X) \}$. For a set S, $\bigsqcup S$ is its least upper-bound.

As explained above, the top of the lattice of monotonic predicate transformers is **miracle**, an infeasible program that is not implementable. Some semantic models do not include this program, or any program that may behave like **miracle** in some situations. In this case, it is usual that, instead of a complete lattice, the semantic model is a CPO: complete partially ordered set.

Definition 2 (Complete partially ordered set). *A* CPO *(complete partially ordered set) is a partially ordered set, which has a bottom, and in which every* directed *subset has a least upper-bound.*

A set is directed if all its finite subsets have an upper bound as one of its own elements. Monotonic functions on a CPO also have fixed points. If, in addition, the function is continuous, then it has a least fixed point as characterised in the theorem below. A function F is continuous if, and only if, it distributes over least upper-bounds of directed sets: $F(\bigsqcup D) = \bigsqcup\{\, d : D \bullet F(d) \,\}$, for every directed set D.

Theorem 2. *For every continuous function $F(X)$ on a CPO with order $\leq$ and bottom element $\bot$*

$$\mu\, X \bullet F(X) = \bigsqcup\{\, n : \mathbb{Z} \bullet F^n(\bot) \,\}$$

For a function F, we define F^0 to be the identity function ($F^0(X) = X$), and $F^n(X) = F(F^{n-1}(X))$, for $n > 0$. Continuity can be a strong property, and is not satisfied by many programs involving unbounded nondeterminism [31]. A more comprehensive account of lattice theory, including proofs for the theorems presented above, can be found in [77].

8 Final Considerations

There are several program and programming models. We have briefly discussed here Hoare logic, refinement algebra, and weakest preconditions. Other models are presented in later chapters of this book. Chapters 3 and 6 give different relational models for the process algebra CSP, and Chapter 4 gives a weakest precondition model for probabilistic programs. Chapter 5 adopts a more operational model (transition systems) and the TLA (Temporal logic of Actions) notation to explore specification, verification and scheduling of real-time and fault-tolerant systems. A common feature to all of them is a formal characterisation of refinement; this is central to any model that supports a development technique.

Potential applications of algebraic reasoning in the context of object-oriented programming is the major objective of the next chapter. Laws of the process algebra CSP are explored in Chapter 3. Laws of programming involving probability are explored in Chapter 4.

Typically, the use of refinement techniques is potentially a very error-prone activity that involves copious formula manipulations. If the benefits of the use of a rigorous programming approach are not to be lost, and are to be available for large-scale industrial systems, the use of tools is essential. Many products are available. Model checking techniques have been particularly attractive to industry due to their high level of automation; they are discussed in detail in Chapter 8. A set of tools that have been very successful in industry for the verification of control system in the area of avionics is presented in Chapter 7.

Transformation Laws for Sequential Object-Oriented Programming

Augusto Sampaio and Paulo Borba

Centro de Informática
Universidade Federal de Pernambuco
Recife - PE, Brazil

In this chapter, we present algebraic laws for a language similar to a subset of sequential Java that includes inheritance, recursive classes, dynamic binding, access control, type tests and casts, assignment, but no sharing. We show that these laws are complete, in the sense that they are sufficient to reduce any program to a normal form substantially close to an imperative program: classes and inheritance are used only to preserve the notion of subtyping; all classes have empty bodies, except the **object** class, which collects all the attributes moved up from all its subclasses. Methods are also eliminated by first resolving dynamic binding, and then in-lining their bodies in place of the calls. This suggests that our laws are expressive enough to formally derive behaviour preserving program transformations; this is illustrated through the derivation of refactorings.

We present the motivation for our work in Section 1. In Section 2, we give an overview of the subset of Java that we consider. After that, in Section 3, we present the algebraic laws of our language, concentrating on its object-oriented features. Completeness of our set of laws is considered in Section 4, where we present the normal form and a reduction strategy. In Section 5 we show how the presented laws can serve as a basis for proving refactorings. Section 6 summarises our results, briefly discusses soundness of the laws, considers the impact of reference semantics, relates our results with work involving concurrency, and suggests topics for further research.

1 Introduction

Reasoning about programs requires the characterisation of the *behaviour* (or *meaning*) of programming operators. As explored in the previous chapter, this is achieved through a formal semantics for the relevant programming language. Program transformations should be based on a notion of equivalence (or refinement) between programs defined using the semantics; they are useful to restructure programs or to support stepwise development from specifications.

There are three major consolidated approaches for defining semantics of programming languages. Broadly, in the operational style [213] an abstract mathematical model of a machine is defined, and the meaning of a program is given in terms of the step by step execution of the program in this model. This allows, for example, to check for feasibility (implementability) of the language operators.

A. Cavalcanti, A. Sampaio, and J. Woodcock (Eds.): PSSE 2004, LNCS 3167, pp. 18–63, 2006.

Another classical approach, the denotational style [233], maps each programming construct to a value (denotation) in some convenient and independent mathematical domain. The use of an explicit mathematical model helps to ensure consistency of the semantic mapping. The weakest precondition semantics discussed in Chapter 1 is an example of a semantics that follows the denotational style: it maps programs to functions from predicates to predicates.

Finally, the algebraic approach is based on postulating general properties of the language constructs, typically as equations that relate the language operators. Unlike the previous approaches, no explicit mathematical model is defined in an algebraic presentation. In principle, this has the advantage of modularity: each new programming notation is introduced with its algebraic properties, hopefully with no effect on the semantics of the existing constructs. Further, equational reasoning, which can be easily mechanised by term rewriting, is immediately available as a framework for reasoning and transforming programs.

Section 6 of the previous chapter has illustrated some algebraic laws of a simple imperative programming language, as well as their use in program transformation and refinement. A set of laws that captures general properties of (and the relationship among) programming constructs is usually called a *refinement algebra*; when the purpose of the laws is to derive executable programs from specifications, then they are collectively denoted as a *refinement calculus*. Our major objective in this chapter is to define a refinement algebra for sequential object-oriented programming, and discuss potential applications of algebraic reasoning in the context of object-orientation.

Several paradigms have benefitted from algebraic programming laws. The laws of imperative programming [116] have been useful for providing algebraic semantic definitions and for establishing a sound basis for formal software development methods. The laws of OCCAM [226] exhibit useful properties of concurrency and communication; similar laws are available for CSP [116, 226]. Algebraic properties of functional programming are elegantly addressed in [28]. Algebraic reasoning for logic programming is presented in [237]. Apart from being useful for reasoning about programs, algebraic reasoning has been proved to be promising for applications such as designing correct compilers [128, 228] and hardware/software partitioning algorithms [242]. As previous contributions related to laws of object-oriented programming, we single out laws for small-grain object-oriented constructs [187, 151]. Some laws have been informally discussed as refactorings [94], and formalised to the degree that they can be encoded in tools [208, 224], but not proved sound or complete.

Laws for an object-oriented programming language similar to sequential Java, but with copy (rather than reference) semantics have been explored in [32, 33]. In [33], laws for the imperative features of the language are presented, but the emphasis is on laws for object-oriented features. The set of laws is shown to be *complete* in the sense that it is sufficient to reduce any program to a normal form close to an imperative program. The formal derivation of refactorings from the proposed set of laws is also illustrated. These results are extended to address soundness of the laws and more refactorings in [32].

In this chapter, we use a uniform example to introduce the language considered in [33, 32], to illustrate the application of some algebraic laws, to derive a design pattern and a refactoring, and to illustrate the completeness result through a normal form reduction process. In addition, we also explore some new issues. The validity of the laws in the context of a language with reference semantics is discussed in some detail, further examples of refinement of class hierarchies are given, and the results are related with other programming paradigms, particularly concurrency. On the other hand, as our major objective is to emphasise the algebraic style, soundness is only briefly addressed in the final section.

Despite our focus on the algebraic approach, by no means we intend to induce a general superiority of the algebraic approach compared to the other presentation styles. As discussed above, each approach serves some noble and distinguishing purposes; the three styles complement each other. For instance, the validity (*soundness*) of a postulated set of laws can be established by linking the algebraic presentation to a denotational or operational model in which the laws can be proved.

Our view, nevertheless, is that once the laws have been proved, in whatever model, they should serve as tools for carrying out program transformations. The mathematical definitions that allow their correctness proofs are normally more complex, and therefore not appealing to practical use. This unified view of semantic presentations is nicely described in detail in [117], which explores the roles of algebras and models in the construction of theories relevant to Computing. This approach to unifying theories of programming has been proposed to study different paradigms, considering a variety of semantic presentations in an integrated way: denotational, operational, and algebraic. Chapter 6 discusses the unifying theories of programming in some detail.

2 The Language

In order to explore in detail the techniques and applications of algebraic reasoning for object-oriented programming, we use a specific language. It essentially is a subset of sequential Java [102] with two major differences:

- it has a copy semantics for both objects and elements of primitive types, whereas Java has a reference semantics for objects;
- it supports both programs and specifications, by also having non-executable constructs, such as specification statements from Morgan's refinement calculus [192].

Similarly to Java, it includes classes, inheritance, access control, dynamic binding, type tests and casts, recursion, assignment, and many other imperative features.

A program $cds \bullet c$ in this language is a set of class declarations cds followed by a main command c, which creates and manipulates objects of the classes in cds. This command might have free variables, corresponding to the program inputs and outputs, but we assume that those can only hold elements of primitive types, not objects. Before discussing class declarations and commands, we further examine the differences just mentioned between our language and Java.

2.1 Copy Semantics

As in the case of Java, our language supports copy semantics for elements of primitive types. However, contrasting with Java, we adopt copy semantics for objects too. So, variables store objects, not references to objects. For variables a and b of a class *Account*, for example, the execution of the assignment $b := a$ stores in b a fresh copy (clone) of the object stored in a. This assignment actually creates a new object in the heap. So changes to the account in a do not affect the account in b, and vice-versa. For example, assuming that the initial balance of a is 0, after executing the following code, the object in a will have balance 600, whereas the object in b will have balance 550.

$$a.credit(550);\ b := a;\ a.credit(50)$$

The method *credit* of *Account*, as expected, increases the balance of an account by the amount it takes as argument. Similarly, object copies, not references, are passed as parameters and returned as results of method calls.

Copy semantics is precisely what is necessary in several situations, and simplifies reasoning by avoiding aliasing and corresponding side-effects. On the other hand, this prevents us from modelling references and sharing. This is necessary, for instance, to implement design patterns such as the Observer [96], where objects mutually refer to each other. In this case, a reference to an object would be stored both in a variable of the main program and in an attribute of another object. In our language, classes can be mutually dependent, being defined in terms of each other, but objects cannot refer to each other.

A detailed discussion of the impact of reference semantics on the set of laws introduced in this chapter is presented in the final section of this chapter.

2.2 Class Declarations

Classes are declared as in the example in Figure 1, where we define a class called *Account*. Subclassing and single inheritance are supported through the **extends** clause. The built-in **object** class is a superclass of all the other classes, so the **extends** clause in the example could have been omitted.

The class *Account* includes three private attributes: *number*, *balance* and *owner*, of types **string**, **double** and *Client*, a user-defined class omitted here. Besides the **pri** qualifier for private attributes, there are qualifiers for protected (**prot**) and public (**pub**) attributes as in Java. For simplicity, the language supports no attribute redefinition and allows only public methods, which can have value and result parameters. The list of parameters of a method is separated from its body by the symbol •. The method *getNumber* has a result parameter n, and *setNumber* has a value parameter also called n. Constructors are declared by the **new** clause and do not have parameters. In contrast to Java, our language adopts a simple semantics for constructors: they are syntactic sugar for methods that are called after creating objects of the corresponding class.

```
class Account extends object
  pri number : string
  pri balance : double
  pri owner : Client
  meth credit ≙ (val value : double • self.balance := self.balance + value)
  meth getBalance ≙ (res bal : double • bal := self.balance)
  meth getNumber ≙ (res n : string • n := self.number)
  meth setNumber ≙ (val n : string • self.number := n)
  meth getOwnerName ≙ (res n : string • self.owner.getName(n))
  new ≙ self.balance := 0; self.owner := new Client
end
```

Fig. 1. Class *Account*

Data types are either primitive (**bool**, **double**, **string** and others) or classes. We consider that methods cannot be mutually recursive, but classes can. However, since we do not support references, we cannot have objects mutually referring to each other.

For simplicity, the language does not have interfaces or abstract classes. However, by defining a method as **abort** we basically specify that nothing can be assumed about how subtypes implement (redefine) that method. In this respect, such a method would be semantically similar to abstract methods in Java.

2.3 Commands

The body of methods and constructors are commands similar to those of Morgan's refinement calculus [192]. Their syntax is formalised as follows.

$c \in Com ::= le := e \mid c;\ c$	assignment, sequence
$\mid x : [\psi_1, \psi_2]$	specification statement
$\mid pc(e)$	parametrised command application
$\mid$ **if** $[\!] i \bullet \psi_i \rightarrow c_i$ **fi**	conditional
$\mid \mu\ X \bullet c \mid X$	recursion, recursive call
$\mid$ **var** $x : T \bullet c$	local variable block
$\mid$ **avar** $x : T \bullet c$	angelic variable block

We allow x, e, le, and T to also denote lists of identifiers, expressions and types; this shall be clear from the context. The expressions le that are allowed to appear as the target of assignments and method calls, and as result arguments, define the subset Le (*left expressions*) of valid expressions. We define this set later in this section.

Method bodies (such as **res** n : **string** • $n :=$ **self**.*number*) are parametrised commands [14, 50], which can be applied to a list of arguments to yield a command, as indicated by the entry $pc(e)$ in the description of commands. So methods are seen as parametrised commands and, therefore, method calls are

represented as the application of parametrised commands. The syntax of parametrised commands is defined as follows.

$$
\begin{array}{lll}
pc \in PCom ::= pds \bullet\ c & & \text{parametrisation} \\
\quad \mid le.m \mid ((N)le).m & & \text{method calls} \\
\quad \mid \mathbf{self}.m \mid \mathbf{super}.m & & \\
pds \in Pds\ \ ::= \varnothing \mid pd \mid pd;\ pds & & \text{parameter declarations} \\
pd \in Pd\ \ \ ::= \mathbf{val}\ x : T \mid \mathbf{res}\ x : T &
\end{array}
$$

The parametrised command $pds \bullet\ c$ declares parameters pds used in a command c. The parametrised command $le.m$ is a call to a method m with target object le. Parameters can be passed by value (keyword **val**) or result (**res**). In the body of the *getOwnerName* method of the class *Account*, for instance, we have a call to a method *getName* with target *owner* and argument n. The name of the account's owner will then be stored in n. A call to a method m on the current object must be written as $\mathbf{self}.m$ since references **self** to the current object are not optional in our language; in the case of redefinitions, the method declared by the superclass can be called by writing $\mathbf{super}.m$.

The conditional (alternation) is in the style of the guarded **if** of Dijkstra's language [81]. In the BNF, we use an informal indexed notation for a finite set of guarded commands $\psi_i \rightarrow c_i$ separated by $[\!]$. In programs, it looks like

$$
\begin{array}{l}
\mathbf{if}\ \psi_1 \rightarrow c_1 \\
[\!]\ \psi_2 \rightarrow c_2 \\
\vdots \\
[\!]\ \psi_n \rightarrow c_n \\
\mathbf{fi}
\end{array}
$$

where ψ_i are conditions and c_i are commands. This command aborts when no condition is valid, and is non-deterministic when more than one condition is valid; an arbitrary valid guard is chosen.

We have recursion and variable blocks. Strictly, a recursive method can be expressed in our language using the recursion operator μ (see [156] for the details). Angelic variables, also known as logical variables or logical constants, are similar to standard local variables (which are not automatically initialised), except that their initial values are angelically chosen to make sure that the program in their scope succeeds, if possible at all. For example, in the program fragment

$$\mathbf{avar}\ x : \mathbb{Z} \bullet \{x = 2\};\ \ldots$$

the variable x is assigned value 2 upon (an implicit) initialisation; otherwise, the assumption $\{x = 2\}$ would behave like **abort**. Angelic declarations are not code, but they are useful for specification and reasoning.

2.4 Expressions

Our language includes typical object-oriented expressions:

$e \in Exp ::=$	**self**	'reference' to current object
	$\mid$ **null** $\mid$ **new** N	null 'reference', object creation
	$\mid x \mid f(e)$	variable, built-in application
	$\mid e$ **is** $N \mid (N)e$	type test, type cast
	$\mid e.x \mid (e;\ x : e)$	attribute selection and update

The expressions **self** and **is** have similar semantics to `this` and `instanceof` in Java; and as in Java, the latter does not require exact type matching. We must write **self**.a to access the attribute a of the current class, since as already mentioned **self** is not optional. The update expression $(e_1;\ x : e_2)$ denotes a copy of the object e_1, but with the attribute x mapped to a copy of e_2; this is similar to update of arrays in Morgan's refinement calculus [192]. So, despite its name, the update expression, similarly to the other expressions, has no side-effects; in fact, it creates a new object instead of updating an existing one. Variables can, however, be updated through the execution of commands, as in $o := (o;\ x : e)$, which is semantically equivalent to $o.x := e$, and updates o. Expressions such as **null**.x and (**null**; $x : e$) cannot be successfully evaluated; they yield the special value **error** and lead the commands in which they appear to abort.

The left-expressions are defined as follows.

$$le \in Le \ ::= \ le1 \mid \textbf{self}.le1 \mid ((N)le).le1 \qquad\qquad le1 \in Le1 \ ::= \ x \mid le1.x$$

These are the expressions that can appear as targets of assignments, and as result arguments; they can also appear as targets of method calls, along with **self**, **super**, and cast expressions.

Exercise 1. Using the language introduced in this section, define a *Client* class having *name* and *address* as attributes, and *getName* (see the *Account* class introduced in Section 2.2), *setName*, and *getStreet* as methods. Assume the *Address* class is defined and contains a *getStreet* method.

Exercise 2. Extend the *Account* class (see Section 2.2) with methods *withdraw* and *transfer*. The method *withdraw* should not leave the balance negative, and *transfer* should receive the target account and the amount of money to be transferred, and return the updated target account. Then write the Java expressions

$$c.credit(c.getBalance() * 3)$$

and

$$c.transfer(d, c.getBalance())$$

as commands in our language.

3 Algebraic Laws

The laws in Chapter 1 are presented as context-independent equations, such as

$$(x := x) = \mathbf{skip}$$

This is the usual approach for imperative programming [116, 226], for instance. Such laws are compositional; they can be used, for example, as rewrite rules and program transformations, and can be applied to any part of a program. We can even think of more than one law being applied simultaneously to different fragments of a program. Due to independence of a particular context, these laws are also applicable to *open* programs: to commands occurring anywhere in an individual class or main program, independent of any other possibly existing class, method, or attribute declarations.

The laws we discuss in this section focus on the object-oriented features of our language. These laws are mostly concerned with properties of class declarations and method calls, which are inherently context-dependent, especially when considering class hierarchies. Therefore, the proposed laws need to address context issues. Equivalence of sets of class declarations cds_1 and cds_2 is denoted by $cds_1 =_{cds,c} cds_2$, where cds is a context of class declarations for cds_1 and cds_2, and c is the main command. This is just an abbreviation for the program equivalence $cds_1\, cds \bullet c = cds_2\, cds \bullet c$. When we write $cds_1 =_{cds,c} cds_2$, we assume cds_1 is well-formed if, and only if, cds_2 is well-formed, and that c is well-formed considering both sets of class declarations $cds_1\, cds$ and $cds_2\, cds$.

These laws consider the entire context, and therefore apply to *closed* programs. This may generate a larger number of side conditions, which are, nevertheless, purely syntactic. Furthermore, although the context is captured for each particular law application, this is by no means a requirement that the context be fixed for successive transformations. The first law introduced below allows elimination and introduction of class declarations; thus its application may change the context of a development. If, eventually, a modified context no longer satisfies the conditions of a law previously applied, this does not invalidate the effected transformation; it just means that in the current context the application of the law would not be valid.

Law 17 (class elimination).

$$cds\ cd_1 \bullet c = cds \bullet c$$

provided

(→) *The class declared in cd_1 is not referred to in cds or c;*

(←) *(1) The name of the class declared in cd_1 is distinct from those of all classes declared in cds; (2) the superclass appearing in cd_1 is either* **object** *or declared in cds; (3) and the attribute and method names declared by cd_1 are not declared by its superclasses in cds, except in the case of method redefinitions.* □

We write '(→)' before the first proviso since it is required only for applications of this law from left to right. We also write '(←)', when a proviso is necessary

only for applying a law from right to left, and '$(\leftrightarrow)$' when it is necessary in both directions. This also helps to interpret each law as two behaviour preserving transformations with different provisos.

We now present laws concerned with attribute and method declarations, method calls, and commands in general.

3.1 Attribute Declarations

The first laws we present in this section allow us to change the declaration of attributes. The following law relates protected and public attributes. From left to right, it establishes that a protected attribute can be made public; from right to left, it asserts that a public attribute can be made protected, provided that it is only directly used by the class in which it is declared and its subclasses. This proviso is necessary to guarantee that the law relates well-formed programs.

Law 18 (change visibility: from protected to public).

class C **extends** D
  prot $a : T$; ads
  ops
end
	$=_{cds,c}$	**class** C **extends** D
		  pub $a : T$; ads
		  ops
		end

provided

$(\leftarrow)$ *$B.a$, for any $B \leq C$, appears only in ops and in the subclasses of C in cds.* □

We write **prot** $a : T$; ads to denote an attribute declaration **prot** $a : T$ followed by all declarations in ads, whereas ops stands for declarations of methods and constructors. The notation $B.a$ refers to uses of the name a via expressions whose static type is exactly B, as opposed to any of its subclasses. For example, if we write that $B.a$ does not appear in ops, we mean that ops does not contain any expression such as $e.a$, for any e of type B, strictly. For an implicit context of class declarations, the subclass relation is denoted by $\leq$, where $A \leq B$ indicates that A is a subclass of B. We consider that any class is a subclass of itself.

Our second law relates private and public attributes.

Law 19 (change visibility: from private to public).

class C **extends** D
  pri $a : T$; ads
  ops
end
	$=_{cds,c}$	**class** C **extends** D
		  pub $a : T$; ads
		  ops
		end

provided

$(\leftarrow)$ *$B.a$, for any $B \leq C$, does not appear in cds, c.* □

When applied from right to left, this law makes a public attribute private. For that, the attribute cannot be used anywhere outside the class where it is declared; this is enforced by the proviso.

The law that allows us to change attribute visibility from private to protected, and vice-versa, can be derived from the previous two laws. In fact, many laws can be derived from the basic ones introduced here. Our aim is not to show those derived laws, but to focus on a concise set of basic laws that captures the essence of the language constructs and are, therefore, powerful enough to derive many other laws. For example, instead of having different laws to deal with attributes with different visibilities, we need only laws dealing with public attributes. The corresponding laws for private and protected attributes can be derived from those since protected and private attributes can always be made public by applying Laws 18 and 19.

The following law establishes that we can move a public attribute a from a class C to a superclass B, and vice-versa. To move the attribute up to B, it is required that this does not generate a name conflict: no subclass of B, other than C, can declare an attribute with the same name; our language does not allow attribute redefinition or hiding as in Java. We do not need to worry about a being declared in B itself, as this is not possible: if it were, then C would not be well-formed. We can move a from B to C provided that a is used only as if it were declared in C.

Law 20 (move attribute to superclass).

class B **extends** A
  ads
  ops
end
class C **extends** B
  pub $a : T$; ads'
  ops'
end
	$=_{cds,c}$	**class** B **extends** A
		  pub $a : T$; ads
		  ops
		end
		class C **extends** B
		  ads'
		  ops'
		end

provided

($\rightarrow$) *The attribute name a is not declared by the subclasses of B in cds;*

($\leftarrow$) *$D.a$, for any $D \leq B$ and $D \not\leq C$, does not appear in cds, c, ops, or ops'.* □

The second proviso, according to the special notation $D.a$ previously introduced, precludes an expression such as **self**.a from appearing in *ops*, but does not preclude **self**.$c.a$, for an attribute $c : C$ declared in B. The last expression is valid in *ops* no matter whether a is declared in B or in C.

The following law allows us to change the class type of an attribute to a supertype, and vice-versa.

Law 21 (change attribute type).

class C **extends** D
  pub $a : T$; ads
  ops
end
	$=_{cds,c}$	**class** C **extends** D
		  pub $a : T'$; ads
		  ops
		end

provided

($\leftrightarrow$) *$T \leq T'$ and every non-assignable occurrence of a in expressions of ops, cds and c is cast with T or any subtype of T declared in cds.*

($\leftarrow$) *(1) every expression assigned to a, in ops, cds and c, is of type T or any subtype of T; (2) every use of a as result argument is for a corresponding formal parameter of type T or any subtype of T.* □

Assignable occurrences of identifiers are result arguments and targets of assignments. For instance, in $\mathbf{self}.a := e$ and $le.m(\mathbf{self}.a)$, the occurrences of a are assignable, if the single parameter of m is passed by result. On the other hand, in an assignment $\mathbf{self}.a.x := e$, there is an assignable occurrence of x but not of a. Therefore, a is required to be cast in the previous proviso. The same comment applies to a result argument $\mathbf{self}.a.x$. Occurrences of identifiers as result arguments and targets of assignments are not cast anywhere; like in Java, this is not allowed in our language.

Exercise 3. Using the laws presented so far, derive a law for changing attribute visibility from private to protected, and vice-versa.

3.2 Method Declarations

In this section we give laws related to the declaration of methods. The following law states that we can introduce or remove a trivial method redefinition, which amounts simply to a call to the method in the superclass.

Law 22 (introduce method redefinition).

class B **extends** A
  ads
  meth $m \;\widehat{=}\; pc$
  ops
end
class C **extends** B
  ads'
  ops'
end
	$=_{cds,c}$	**class** B **extends** A
		  ads
		  meth $m \;\widehat{=}\; pc$
		  ops
		end
		class C **extends** B
		  ads'
		  meth $m \;\widehat{=}\;$ **super**.m
		  ops'
		end

provided ($\rightarrow$) *m is not declared in ops'.* □

Strictly, we cannot define a method as **meth** $m \mathrel{\widehat{=}}$ **super**.m. A method declaration is an explicit parametrised command, so that, above, pc has the form $(pds \bullet c)$; the redefinition of m should be **meth** $m \mathrel{\widehat{=}} (pds \bullet \mathbf{super}.m(\alpha pds))$, where αpds denotes the list of parameter names declared in pds. For simplicity, however, we adopt the shorter notation **meth** $m \mathrel{\widehat{=}}$ **super**.m.

The next law states that we can merge a method declaration and its redefinition into a single declaration in the superclass. The resulting method uses type tests to choose the appropriate behaviour.

Law 23 (move redefined method to superclass).

```
class B extends A
  ads
  meth m ≙ (pds • b)
  ops
end
class C extends B
  ads'
  meth m ≙ (pds • b')
  ops'
end
```

$=_{cds,c}$

```
class B extends A
  ads
  meth m ≙ (pds •
   if ¬(self is C) → b
   [] self is C → b'
   fi)
  ops
end
class C extends B
  ads'
  ops'
end
```

provided

($\leftrightarrow$) *(1)* **super** *and private attributes do not appear in* b'*; (2)* **super**.m *does not appear in* ops'*;*

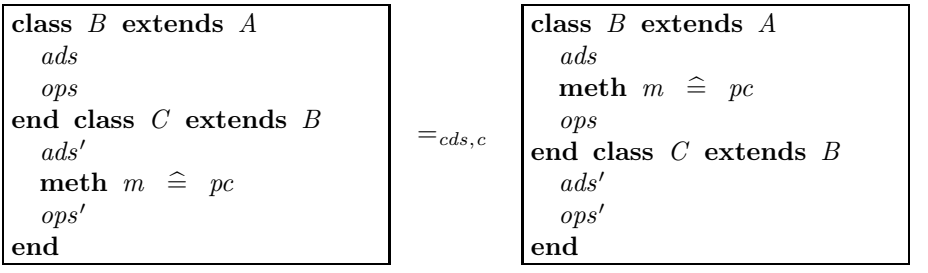
($\rightarrow$) b' *does not contain uncast occurrences of* **self** *nor expressions of the form* $((C)\mathbf{self}).a$ *for any protected attribute* a *in* ads'*;*

($\leftarrow$) m *is not declared in* ops'*.* □

The provisos concerning **super** are needed because its semantics may be affected if it is moved from a subclass to a superclass, or vice-versa. The other provisos ensure the validity of the programs involved. We can only move the body of m up if it does not refer to elements of the class where it is declared through uncast **self**. As mentioned in the previous section, **self** must be used for calling methods and selecting attributes of the current object.

Our third method law allows us to move up in the class hierarchy a method declaration that is not a redefinition. Our language supports method redefinition but, as opposed to Java, not overloading. Hence, we cannot have different methods in the same class, or in a class and a subclass, with the same name, but different parameters. Our law indicates that we can move a method down too, if this method is used only as if it were defined in the subclass.

Law 24 (move original method to superclass).



provided

(↔) *(1)* **super** *and private attributes do not appear in pc; (2) m is not declared in any superclass of B in cds;*

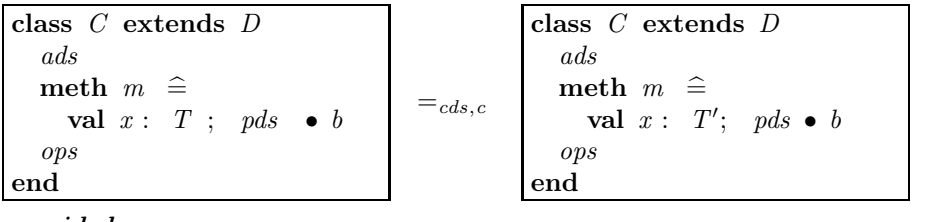
(→) *(1) m is not declared in ops, and can only be declared in a class D, for any* $D \leq B$ *and* $D \not\leq C$*, if it has the same parameters as pc; (2) pc does not contain uncast occurrences of* **self** *nor expressions in the form* $((C)\mathbf{self}).a$ *for any protected attribute a in ads′;*

(←) *(1) m is not declared in ops′; (2) D.m, for any* $D \leq B$ *and* $D \not\leq C$*, does not appear in cds, c, ops or ops′.* □

The provisos for this law are similar to those of Laws 23 and 20. Only the first two are necessary to preserve semantics; the others guarantee that we relate syntactically valid programs. The second proviso, associated to applications of the law in both directions, precludes superclasses of B from defining m, because, otherwise, when moving it, we could affect the semantics of calls such as $b.m(e)$, for a b storing an object of B.

The next two laws allow us to change the type of a parameter; they are similar to Law 21. The first law handles value parameters.

Law 25 (change value parameter type).



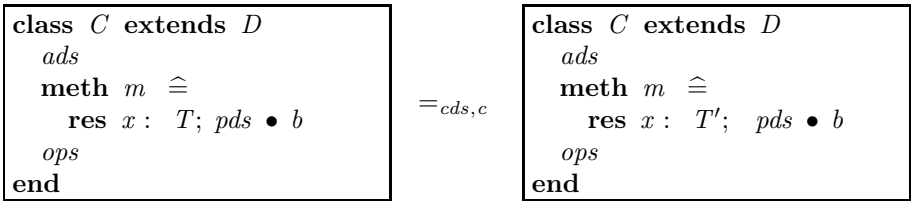
provided

(↔) $T \leq T'$ *and every non-assignable occurrence of x in expressions of b are cast with T or any subtype of T;*

(←) *(1) every actual parameter associated with x in ops, cds and c is of type T or any subtype of it; (2) every expression assigned to x in b is of type T or any subtype of T; (3) every use of x as result argument in b is for a corresponding formal parameter of type T or any subtype of T.* □

For a result parameter, we have the following law. As opposed to a value argument, the type of a result argument has to be that of the corresponding formal parameter or a supertype of it. We cannot change the type of a parameter to a supertype of any of the arguments used in the program.

Law 26 (change result parameter type).



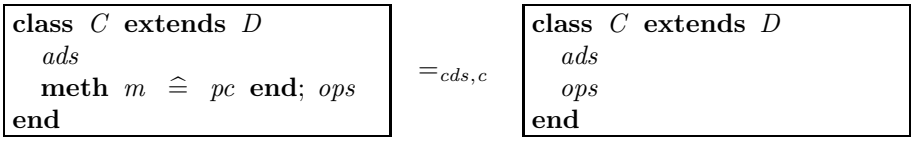
provided

(↔) *$T \leq T'$ and every non-assignable occurrence of x in expressions of b are cast with T or any subtype of T;*
(→) *every actual parameter associated with formal parameter x in ops, cds and c is of type T' or any supertype of it;*
(←) *(1) every expression assigned to x in b is of type T or any subtype of T; (2) every use of x as result argument in b is for a corresponding formal parameter of type T or any subtype of T.* □

The first proviso is the same as that in the previous law: it restricts the way in which the parameter is used in the method body. The second proviso is related to the use of arguments. The third proviso is similar to that in Law 21.

A method that is not called can be eliminated. Conversely, we can always introduce a new method in a class, provided we avoid naming conflicts.

Law 27 (method elimination).



provided

(→) *$B.m$ does not appear in cds, c nor in ops, for any B such that $B \leq C$.*
(←) *m is not declared in ops nor in any superclass or subclass of C in cds.* □

The introduction and elimination of attributes is considered in Section 3.5.

3.3 Method Calls

The laws in this and in the next section give properties of the equivalence relation for commands, instead of programs as those in the previous sections. The following law indicates that we can replace a method call **super**.m in a class C by a copy of the body of m as declared in the immediate superclass of C, provided the body does not contain **super** nor private attributes.

Law 28 (eliminate super). *Consider that CDS is a set of two class declarations as follows.*

class B **extends** A	**class** C **extends** B
ads	ads'
meth $m \mathrel{\widehat{=}} pc$	ops'
ops	**end**
end	

Then we have that cds CDS, $C \rhd$ **super**.$m \;=\; pc$
provided $(\rightarrow)$ **super** *and the private attributes in ads do not appear in pc.* □

The notation *cds CDS* denotes the union of the class declarations in *cds* and *CDS*, and $cds, N \rhd c \;=\; d$ indicates that the equation of commands $c \;=\; d$ holds inside class named N, in a context defined by the set of class declarations *cds*. Instead of a class name, we might use **main** for asserting that the equality holds inside the main program.

Law 28 is similar to the standard copy rule for procedures [192]; for calls **super**.m, dynamic binding does not apply. The arguments to which **super**.m is applied are not touched by this law; *pc* ends up applied to the same arguments.

In the case where a method is not redefined, and there are no visibility concerns, we can again use the copy rule to characterise method calls. It might be surprising that we need only such simple laws to characterise method call elimination. The reason is that dynamic binding is handled by Law 23 as a separate issue. Hereafter, the notation $cds, N \rhd e : C$ is used to indicate that in the class N declared in *cds*, the expression e has static type C. Again, instead of a class name, we might use **main** for asserting that the typing holds inside the main program.

Law 29 (method call elimination). *Consider that the following class declaration*

class C **extends** D
  ads
  meth $m \mathrel{\widehat{=}} pc$
  ops
end

is included in cds, and that $cds, A \rhd le : C$. *Then*

$$cds, A \rhd le.m(e) \;=\; \{le \neq \mathbf{null} \wedge le \neq \mathbf{error}\};\; pc[le/\mathbf{self}](e)$$

provided

$(\leftrightarrow)$ *(1) m is not redefined in cds and pc does not contain references to* **super***;*
(2) all attributes which appear in the body pc of m are public. □

A method call $le.m(e)$ aborts when le is **null** or **error**. Thus, we need the assumption $\{le \neq \mathbf{null} \wedge le \neq \mathbf{error}\}$ on the right-hand side of the law. The law for a call $\mathbf{self}.m(e)$ is similar. As already said in Section 2, the assumption $\{b\}$ behaves like **skip** if b is true, and as **abort** otherwise. The notation $pc[le/\mathbf{self}]$ denotes the parametrised command pc where **self** is replaced with le.

A type cast plays two major roles. At compilation time, casting is necessary when using an expression in contexts where an object value of a given type is expected, and this type is a strict subtype of the expression type. For example, if $x : B$, $C \leq B$ and a is an attribute which is in C but not in B, then the selection of this attribute using x requires a cast, as in $((C)x).a$. If a is declared in B, then the cast is not necessary for compilation, but once it is there, it cannot simply be eliminated, because a cast also has a run time effect.

At run time, if the value of a cast expression does not have the required type, its evaluation results in **error**, and the command in which it appears aborts. In the previous example, if the attribute a is in class B, although the cast could be eliminated regarding its static effect, it still has a dynamic effect when the object value of x happens to be of type B, but not of type C.

In order to capture the behaviour of casts, we use assumptions. The following law deals with the elimination of type casts in targets of method calls.

Law 30 (eliminate cast: method call). *If* $cds, A \rhd e : B$, $C \leq B$ *and* m *is declared in* B *or in any of its superclasses in* cds, *then*

$$cds, A \rhd ((C)e).m(e') \;=\; \{e \;\mathbf{is}\; C\};\; e.m(e') \qquad \square$$

Casts in arguments can also be eliminated, but we omit this similar law.

3.4 Commands and Expressions

In the same way that the type of an attribute (Law 21) or parameter (Laws 25 and 26) can be changed if all its uses are cast, we can also change the type of a local variable in this case.

Law 31 (change variable type).

$$cds, A \rhd \mathbf{var}\; x : T \bullet c \;=\; \mathbf{var}\; x : T' \bullet c$$

($\hookrightarrow$) $T \leq T'$ *and every non-assignable occurrence of* x *in expressions of* c *is cast with* T *or any subtype of* T;

($\hookleftarrow$) *(1) every expression assigned to* x *in* c *is of type* T *or any subtype of* T; *(2) every use of* x *as result argument in* c *is for a corresponding formal parameter of type* T *or any subtype of* T. $\square$

The same holds for angelic variables.

The following law formalises the fact that any expression can be cast with its declared type.

Law 32 (introduce trivial cast in expression). *If* $cds, A \rhd e : C$, *then*

$$cds, A \rhd e \;=\; (C)e \qquad \square$$

For simplicity, this is formalised as a law of expressions, not commands. Nevertheless, it should be considered as an abbreviation for several laws of assignments, conditionals, and method calls that deal with each possible pattern of expressions. For example, it abbreviates the following laws, and many others, all with the same proviso as Law 32.

$$cds, A \rhd le := e.x \;=\; le := ((C)e).x$$
$$cds, A \rhd le.m(e) \;=\; le.m((C)e)$$
$$cds, A \rhd le.m(e) \;=\; ((C)le).m(e)$$

As illustrated by the last example, this law is equally valid for left-expressions, which are a form of expression. However, our language, like Java, does not allow casts to appear in targets of assignments and result parameters. So Law 32 should not be considered an abbreviation for equations such as

$$cds, A \rhd e := e'.x \;=\; ((C)e) := e'.x$$

which are not valid since $((C)e) := e'.x$ is not a command in our language, even if e is a left expression.

Law 30, presented in the previous section, allows us to eliminate casts in targets of method calls. We can also eliminate casts in assignments using the law below.

Law 33 (eliminate cast: expression). *If* $cds, A \rhd le : B$ *and* $cds, A \rhd e : B'$, *with* $C \leq B'$ *and* $B' \leq B$, *then*

$$cds, A \rhd le := (C)e \;=\; \{e \textbf{ is } C\};\ le := e$$ □

Similar laws apply to expressions in conditionals and other points of a program.

Two simple laws of type test are presented below; they are laws of expressions. Law 34 asserts that the type test **self is** M is true when appearing inside any subclass of M, including M itself, of course.

Law 34 (is test true). *If* $N \leq M$, *then* $cds, N \rhd \textbf{self is } M = \textbf{true}$ □

In complement to Law 34, the following law asserts that type tests for unrelated classes are mutually exclusive.

Law 35 (is test exclusive). *If* $N \not\leq M$ *and* $M \not\leq N$, *then*

$$cds, C \rhd \textbf{self is } M = \textbf{self is } M \wedge \neg(\textbf{self is } N)$$ □

Similarly, we have laws expressing basic properties of **self**.

Law 36 (self and null). $cds, C \rhd \textbf{self} \neq \textbf{null} = \textbf{true}$ □

Law 37 (self and error). $cds, C \rhd \textbf{self} \neq \textbf{error} = \textbf{true}$ □

The law for attribute access establishes when such an access is successful.

Law 38 (attribute access).

$$cds, C \rhd le.a \neq \textbf{error} = le \neq \textbf{null} \ \wedge le \neq \textbf{error}$$ □

The following laws express simple properties of the alternation command. The first law allows us to simplify an alternation whose commands are the same in all branches, assuming that the disjunction of all guards is true.

Law 39 (if identical guarded commands). *If* $\bigvee i : 1 \mathinner{..} n \bullet \psi_i = \mathbf{true}$*, then*

$$cds, C \rhd \mathbf{if} \ [\!] \ i : 1 \mathinner{..} n \bullet \psi_i \rightarrow c \ \mathbf{fi} \ = \ c. \qquad \square$$

The other law states that the order of the guarded commands of an alternation is immaterial.

Law 40 (if symmetry). *If* π *is any permutation of* $1 \mathinner{..} n$*, then*

$$cds, C \rhd \mathbf{if} \ [\!] \ i : 1 \mathinner{..} n \bullet \psi_i \rightarrow c_i \ \mathbf{fi} \ = \ \mathbf{if} \ [\!] \ i : 1 \mathinner{..} n \bullet \psi_{\pi(i)} \rightarrow c_{\pi(i)} \ \mathbf{fi} \ \square$$

Many other command and expression laws [116, 192] are useful for reasoning about the imperative features of our language, but we omit them here since our focus is on the object-oriented features.

Exercise 4. Using the laws presented in this section, derive the following law that asserts that the test **self is** M is false inside a class N, provided N is not a subclass of M, and vice-versa.

Law 41 (is test false). *If* $N \not\leq_{cds} M$ *and* $M \not\leq_{cds} N$*, then*

$$cds, N \rhd \mathbf{self\ is}\ M = \mathbf{false} \qquad \square$$

Exercise 5. Consider the following class declarations.

```
class B
  pri a : ℤ
  pri b : double
  meth m ≙ if self is C → self.a := 3 [] self is D → self.b := 3.25 fi
end
class C extends B end
class D extends B end
```

Derive an equivalent set of class declarations where the attributes a and b are respectively declared in C and D. Are those set of declarations equivalent in all contexts? Use the presented laws and the following laws for conditionals.

Law 42 (abort conditional).

$$cds, C \rhd \begin{pmatrix} \mathbf{if} \ \ \psi \rightarrow c \ [\!] \ \psi' \rightarrow c' \ \mathbf{fi} \\ = \mathbf{if} \ \ \psi \rightarrow c \ [\!] \ \psi' \rightarrow c' \ [\!] \ \neg\psi \wedge \neg\psi' \rightarrow \mathbf{abort} \ \mathbf{fi} \end{pmatrix} \qquad \square$$

Law 43 (nest conditional).

$$cds, C \rhd \begin{pmatrix} \mathbf{if} \ \ \psi \wedge \psi_1 \rightarrow c_1 \ [\!] \ \psi \wedge \psi_2 \rightarrow c_2 \ [\!] \ \psi' \rightarrow c' \ \mathbf{fi} \\ = \mathbf{if} \ \ \psi \rightarrow (\mathbf{if} \ \ \psi_1 \rightarrow c_1 \ [\!] \ \psi_2 \rightarrow c_2 \ \mathbf{fi}) \ [\!] \ \psi' \rightarrow c' \ \mathbf{fi} \end{pmatrix} \qquad \square$$

Exercise 6. Apply some of the laws presented so far in order to obtain a program that is equivalent to

$$\begin{array}{l}\textbf{var } a, b : \mathit{Account} \bullet \\ \quad a := \textbf{new}\mathit{Account};\ b := a;\ a.\mathit{credit}(5);\ a.\mathit{transfer}(2, b, b);\ \ldots\end{array}$$

but that does not have any casts and uses only variables of type **object**.

3.5 Class Refinement

Besides the equivalence laws presented so far, reasoning about classes usually requires a notion of class refinement, which is related to data refinement. This is necessary, for example, to show that the following class declaration

$$\begin{array}{l}\textbf{class } \mathit{Flag} \\ \quad \textbf{pri } b : \textbf{bool} \\ \quad \textbf{new} \mathrel{\widehat{=}} \textbf{self}.b := \textbf{true} \\ \quad \textbf{meth } \mathit{change} \mathrel{\widehat{=}} \textbf{self}.b := \mathit{not}(\textbf{self}.b) \\ \quad \textbf{meth } \mathit{get} \mathrel{\widehat{=}} (\textbf{res } r : \mathit{boolean} \bullet r := b) \\ \textbf{end}\end{array}$$

which uses a boolean attribute to record its current status, can be replaced by the declaration

$$\begin{array}{l}\textbf{class } \mathit{Flag} \\ \quad \textbf{pri } i : \mathbb{Z} \\ \quad \textbf{new} \mathrel{\widehat{=}} \textbf{self}.i := 1 \\ \quad \textbf{meth } \mathit{change} \mathrel{\widehat{=}} \textbf{self}.i := \mathit{rem}(\textbf{self}.i + 1, 2) \\ \quad \textbf{meth } \mathit{get} \mathrel{\widehat{=}} (\textbf{res } r : \mathit{boolean} \bullet r := \textbf{self}.i = 1) \\ \textbf{end}\end{array}$$

which internally uses an integer (with `1` playing the rôle of **true** and `0` of **false**), but offers the same observational behaviour as the previous one. Users of *Flag* through its *change* and *get* methods would not notice if the first declaration were replaced by the second. So, we say that the first declaration of *Flag* is refined by the second. In this particular case, we also have that the second declaration is refined by the first. The declarations are actually equivalent. This kind of equivalence, however, cannot be proved with the laws of classes previously introduced. When a change of data representation, as from **bool** to $\mathbb{Z}$ in class *Flag*, is involved, we have to resort to different notions of refinement and equivalence and their associated laws.

The equivalence notion of class declarations used so far, $\mathit{cdsa} =_{\mathit{cds},c} \mathit{cdsc}$, is just an abbreviation for the program equivalence $\mathit{cdsa}\ \mathit{cds} \bullet c = \mathit{cdsc}\ \mathit{cds} \bullet c$, which simply corresponds to program refinement in both directions:

$$\mathit{cdsa}\ \mathit{cds} \bullet c \sqsubseteq \mathit{cdsc}\ \mathit{cds} \bullet c \text{ and } \mathit{cdsc}\ \mathit{cds} \bullet c \sqsubseteq \mathit{cdsa}\ \mathit{cds} \bullet c$$

Similarly, $\mathit{cdsa} =_{\mathit{cds},c} \mathit{cdsc}$ corresponds to $\mathit{cdsa} \sqsubseteq_{\mathit{cds},c} \mathit{cdsc}$ and $\mathit{cdsc} \sqsubseteq_{\mathit{cds},c} \mathit{cdsa}$, which are abbreviations for the program refinements just presented.

We now use a different refinement notion that does not rely on uses of classes by a specific main command c [49]. This is called class refinement and is defined as follows. We say that a sequence of abstract class declarations *cdsa* is refined by concrete class declarations *cdsc*, in the context of class declaration *cds*, which is written $cdsa \preccurlyeq_{cds} cdsc$ if, and only if, the following two conditions hold.

(a) *cdsa cds* and *cdsc cds* are both well-formed;
(b) for all commands c that use only methods and public attributes in *cds* and *cdsa*, if c is well-typed for *cdsa cds*, then c is also well-typed in *cdsc cds*; and $cdsa\ cds \bullet c \sqsubseteq cdsc\ cds \bullet c$.

By definition, proving that $cdsa \preccurlyeq_{cds} cdsc$ guarantees that $cdsa \sqsubseteq_{cds,c} cdsc$ holds for all commands c satisfying the constraints above. This reveals the intuition behind the definition: the classes in *cdsa* can be replaced by the classes in *cdsc* without affecting the behaviour of main commands that use those classes through their methods and public attributes only. It does not matter if the classes in *cdsa* have "abstract" attributes that are implemented by different "concrete" ones in *cdsc*; we are just interested on the external behaviour of those classes, not how they internally represent state.

We can now manipulate class declarations in isolation. We do not need to mention main commands as in several laws introduced so far. Proving class refinement, however, involves nontrivial induction on commands. To avoid that, we rely on simulation laws that entail class refinement [49]. The simulation relation $cdas \preceq_{cds,CI} cdsc$ is similar to class refinement, but takes an additional argument: a coupling invariant *CI* that relates attributes in *cdsa* with corresponding ones in *cdsc*. The fact that simulation entails class refinement is formalised as $cdas \preceq_{cds,CI} cdsc$, for a coupling invariant *CI*, implies $cdsa \preccurlyeq_{cds\ cds'} cdsc$ for any cds' not containing subclasses of *cdsa*.

Traditional simulation laws for data refinement deal with modules that encapsulate variables, since this is required for changing the internal representation of those variables. Our first law follows this approach. It allows us to change private attributes in a class, relating them with already existing attributes by means of a coupling invariant. The application of this law changes the bodies of the methods declared in the class. The changes follow the traditional laws for data refinement [192].

Law 44 (pri attribute-coupling invariant).

class A **extends** C
  $adsA$; ads
  ops
end
	$\preceq_{cds,CI}$	**class** A **extends** C
		  $adsC$; ads
		  $CI(ops)$
		end

provided *adsA and adsC contain only private attributes.* □

By convention, the attributes denoted by *adsA* are abstract, whereas those denoted by *adsC* are concrete. The coupling invariant *CI* relates abstract and

concrete attributes. The notation $CI(ops)$ indicates that the methods in *ops* are transformed according to command laws of data refinement [192] using CI as a coupling invariant: every guard may assume the coupling invariant and every command is extended by modifications to the new variables so that the coupling invariant is maintained.

The law below, which can be used to introduce and eliminate attributes, is a direct application of the previous law and definitions.

Law 45 (attribute elimination).

class B **extends** A
  pri $a : T$; ads
  ops
end
	$=_{cds,c}$	**class** B **extends** A
		  ads
		  ops
		end

provided

($\rightarrow$) *$B.a$ does not appear in ops;*
($\leftarrow$) *a does not appear in ads and is not declared as an attribute by a superclass or subclass of B in cds.* □

If a private attribute is not in use inside the class in which it is declared, we can remove it. This law can be proved from the definition of class refinement and with two applications of Law 44, for obtaining refinement in both directions. From left to right, for example, a should be regarded as abstract, there should be no concrete attributes, and the coupling invariant should be true.

Going beyond traditional simulation laws, we now consider changing data representation in a hierarchy of classes. The simulation Law 46 below allows us to change protected attributes in a class, relating them with already existing attributes by means of a coupling invariant. The application of this law changes the bodies of the methods declared in the class and in its subclasses, since the refined protected attributes might be used there as well. In fact, simulation in this case assumes a notion of module that extends the boundaries of classes to consider its subclasses too, according to the visibility policy defined by **prot**.

Law 46 (prot attribute-coupling invariant).

class A **extends** C
  $adsA$ ads
  ops
end
cds'
	$\preceq_{cds,CI}$	**class** A **extends** C
		  $adsC$ ads
		  $CI(ops)$
		end
		$CI(cds')$

provided

($\leftrightarrow$) *(1) $adsA$ and $adsC$ contain only protected attributes; (2) CI does not refer to private attributes in ads; (3) cds' only contains subclasses of A; (4) cds contains no subclasses of A.* □

The notation $CI(cds')$ indicates that all methods in cds' are transformed according to the command laws of data refinement. These transformations are also done in the class A; as before, this is indicated by the notation $CI(ops)$.

These simulation laws, together with laws of commands and of the object-oriented features, form a solid basis for proving more elaborate transformations of object-oriented programs, as illustrated in Section 5 on formal refactoring.

Exercise 7. In order to derive Law 45, we need two applications of Law 44 to establish class refinement in both directions. Choose an appropriate coupling invariant and prove class refinement from right to left.

Exercise 8. Using $i = 1 \Leftrightarrow b = \mathbf{true} \wedge i = 0 \Leftrightarrow b = \mathbf{false}$ as coupling invariant, prove that the first declaration of *Flag* at the beginning of the section is refined by the second. Prove class refinement in the other direction too.

4 Completeness

An algebraic characterisation of the semantics of a programming language (or of an abstract data type) immediately raises the issue of *completeness*: is it possible to derive any valid property from the equations (laws)? As we know from Gödel's incompleteness theorem (see, for instance, [250]), there is no consistent and finite set of axioms that completely characterises even elementary arithmetic, let alone a programming language like the one introduced in Section 2.

On the other hand, it is important to study the comprehensiveness of a proposed set of laws. A standard approach is to define a reduction strategy, based on the laws, whose target is a normal form described in terms of a restricted subset of the language being discussed. This shows that the laws are sufficiently powerful to reduce any program to this normal form. This is what we describe here, for a normal form that uses classes and inheritance only to preserve the notion of subtyping; all classes have empty bodies, except **object**, which may include attribute declarations. This suggests that our laws are expressive enough to reason about the object-oriented structure of programs.

We consider that a program $cds \bullet c$ is in subtype normal form if it obeys the following conditions.

- Each class declaration in cds, except **object**, has an empty body;
- The **object** class declares only public attributes, and their types are either primitive or **object**;
- All local declarations in the main command c are of variables whose types are either primitive or **object**;
- No type cast is used in c.

In a program in subtype normal form, the class **object** is explicitly included. All other classes may include the inheritance and subtype clause **extends**, but no declaration of methods, constructors or attributes is allowed. As an example

of the structure of a program in normal form, consider a simple bank account application whose class diagram is depicted in Figure 2(a); the structure of the corresponding normal form is given in Figure 3(g), also as a class diagram. Regarding the actual code of the bank account example, it is collected in Figure 4, and its normal form program in Figure 10. For conciseness, here we simplify the class *Account*, introduced in Section 2, by removing the attribute *owner*, the method *getOwnerName* and the class constructor. The class *BonusAccount* extends *Account* by redefining the method *Credit* to generate a bonus: a percentage of the amount credited in the account; the bonus is recorded in the attribute *bonus* and can be inspected through the method *getBonus*. The class *SavingAccount* also inherits from account; it simply adds a new method to compute interest. There is also the class *AccountList* that plays the role of a collection of accounts. It is a recursive class: the attribute *acc* stores an account and *prox* is itself a list of accounts.

Although this normal form preserves object-oriented features, namely the subtype hierarchy, object creation, and type tests, it is substantially close to an imperative program. The class **object**, the only one with explicitly declared elements, takes the form of a recursive record, as it contains only public attributes. As no methods are allowed, the main command c is similar to an imperative program, even though object creation and type test can still be used.

For the elimination of all object-oriented features, the natural normal form is the imperative subset of our language extended with recursive records. A reduction to such a form, which would yield a stronger completeness result, requires some sort of mapping from an object to a relational model; an extra variable is necessary to keep the type information. The subtype normal form, however, is close to an imperative program, and some of the additional laws for a reduction to a pure imperative program are presented in Section 3.5.

It is important to note that the reduction of a program to normal form does not suggest a compilation process. Its sole purpose is to show that we have a comprehensive set of laws, which can be used to yield an equivalent program written with a small subset of constructs. Roughly, the fewer constructs, the better the normal form; a similar approach has been used for other programming paradigms [116, 226].

The reduction strategy involves the following major steps:

- Move all the attribute and method declarations in the classes of the original program to the **object** class;
- Change all the declarations of class-typed identifiers to define their type to be **object**;
- Eliminate casts;
- Eliminate method calls and declarations.

Before analysing each step in detail, we illustrate the overall reduction strategy in terms of class diagrams. Considering the diagram in Figure 2(a) as a representation of the original program, the first relevant transformation is to move

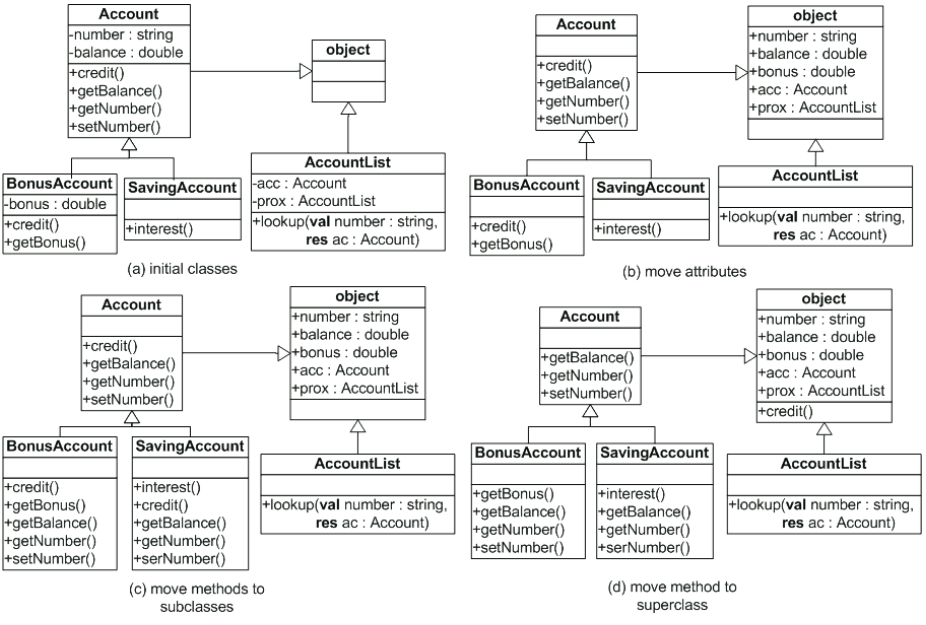


Fig. 2. Bank account diagrams—attributes and method credit in **object**

attributes up to the **object** class, as presented in Figure 2(b). Note that, in this process, attribute visibility is changed to **pub** (represented as a + signal in the diagram, whereas - precedes **pri** attributes). The reason for changing visibility is that later on methods are eliminated, and direct access to the attributes in the object class becomes necessary.

In order to move methods up to the **object** class, all methods of a class are redefined in the subclasses, if they are not already; this is illustrated in Figure 2(c). The purpose of these (trivial) redefinitions is to simplify moving methods. In Figure 2(d), we single out moving method `credit` to **object**; Figure 3(e) shows the result of moving the remaining methods.

The change of all declarations of class-typed identifiers to type **object** is illustrated in Figure 3(f); observe that declarations of identifiers of primitive types are not affected. The next step, cast elimination, cannot be illustrated through diagrams; this is dealt with and illustrated later on as code transformation. Finally, method declarations are eliminated (Figure 3(g)), and the corresponding bodies are in-lined as replacements of the corresponding calls; this is valid because dynamic binding is resolved when methods are moved up in the hierarchy, and all attributes are public.

In the remainder of this section we present the reduction strategy in detail, as a sequence of simple and incremental steps justified by the application of algebraic laws. We illustrate the process using the actual code of the bank account example summarised in Figure 4, but the process is actually general.

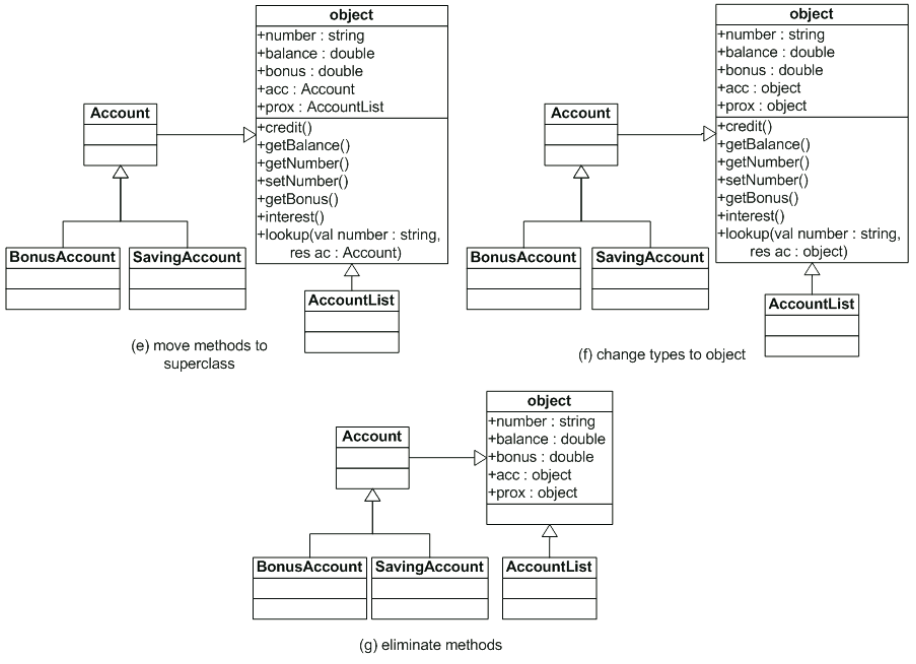


Fig. 3. Bank account diagrams—methods up, change type to **object**, method elimination

4.1 Make Attributes Public

The first major step in our reduction strategy is to move up the attributes. Nonetheless, before that, we need to make sure that they are either public or protected, otherwise method declarations in the subclasses might become invalid. As the final step of the strategy eliminates methods and requires that the main program have direct access to the attributes, we make all attributes public.

For that, we apply two laws: Law 18 to make protected attributes public, and Law 19 to make private attributes public. In the strategy all the laws are applied from left to right. We need to exhaustively apply these two laws to all classes in *cds*. In our example, only Law 19 is effectively applied to classes *Account*, *BonusAccount*, and *AccountList*.

4.2 Move Attributes Up

After changing the visibility of all attributes to public, we move them up to the **object** class using Law 20. Starting from the bottom of the class hierarchy, and moving upwards, the exhaustive application of this law moves all attributes to **object**. We assume that two distinct classes are not allowed to declare attributes with the same name. Therefore, name conflicts do not arise and the proviso of

```
class Account
  pri number : string
  pri balance : double
  meth credit ≙ (val value : double • self.balance := self.balance + value)
  meth getBalance ≙ (res bal : double • bal := self.balance)
  meth getNumber ≙ (res n : string • n := self.number)
  meth setNumber ≙ (val n : string • self.number := n)
end


class BonusAccount extends Account
  pri bonus : double
  meth credit ≙ (val value : double •
    super.credit(value); self.bonus := value * 0.05)
  meth getBonus ≙ (res bon : double • bon := self.bonus)
end


class SavingAccount extends Account
  meth interest ≙
    var bal : double • self.getBalance(bal); self.credit(bal * 0.1)
end


class AccountList
  pri acc : Account
  pri prox : AccountList
  meth lookup ≙ (val number : string; res ac : Account •
    if (self.acc = null) → ac := null
    [] ¬ (self.acc = null) → var n : string •
                               self.acc.getNumber(n);
                               if (n = number) → ac := self.acc
                               [] ¬ (n = number) → self.prox.lookup(number, ac)
                               fi


    fi)
end


• var acc : Account •
  acc := new SavingAccount;
  acc.setNumber("21.342 − 7");
  acc.credit(200);
  if (acc is SavingAccount) → ((SavingAccount)acc).interest()
  [] ¬ (acc is SavingAccount) → skip
  fi
```

Fig. 4. Bank account program

```
class object
  pub number : string; balance : double; acc : Account
  pub prox : AccountList; bonus : double
end
class Account
  meth credit ≙ (val value : double • self.balance := self.balance + value)
  meth getBalance ≙ (res bal : double • bal := self.balance)
  meth getNumber ≙ (res n : string • n := self.number)
  meth setNumber ≙ (val n : string • self.number := n)
end
class BonusAccount extends Account
  meth credit ≙ (val value : double •
    super.credit(value); self.bonus := value ∗ 0.05)
  meth getBonus ≙ (res bon : double • bon := self.bonus)
end
class SavingAccount extends Account
  meth interest ≙
    var bal : double • self.getBalance(bal); self.credit(bal ∗ 0.1)
end
class AccountList
  meth lookup ≙ (val number : string; res ac : Account •
    if (self.acc = null) → ac := null
    [] ¬ (self.acc = null) → var n : string •
                               self.acc.getNumber(n);
                               if (n = number) → ac := self.acc
                               [] ¬ (n = number) → self.prox.lookup(number, ac)
                               fi

    fi)
end
• ...
```

Fig. 5. Bank account program—attributes in **object**

the law is always valid. Our assumption imposes no significant restriction on our approach, since renaming can be used to meet this requirement.

Each attribute is progressively moved to its immediate superclass until it reaches the object class. In our example, *number* and *balance* of *Account* are directly moved to **object**; the same happens with the attributes *acc* and *prox* of *AccountList*. The attribute *bonus* of *BonusAccount* is first moved to *Account* and then to **object**. Part of the bank account program, after this transformation, is presented in Figure 5. The class **object** is explicitly defined to include all the attributes of the original classes, which now do not declare any attributes; the main program is not touched. Part of the object-oriented design is lost, but the program still behaves as before. The purpose here is to establish the expressiveness of the laws; in practical applications of program transformation, like refactoring (see Section 5), the laws are applied in the reverse order.

4.3 Introduce Trivial Method Redefinitions

This and the next step are auxiliary to moving methods up, which requires the elimination of occurrences of **super** in the method bodies. When eliminating **super** there might be an arbitrary number of classes (in the inheritance hierarchy) between the class in which a method is defined and the class which invokes the method via **super**. To avoid dealing with such arbitrary cases, we introduce trivial method redefinitions using **super**. If an inherited method does not have a redefinition, in this step we provide a trivial redefinition that simply calls the method of the superclass.

We exhaustively apply Law 22, from left to right, considering all methods of all classes with subclasses. We start from **object** and move downwards in the class hierarchy. At the end, all classes have a definition for the methods they provide: either a trivial redefinition or that in the original program.

Considering our bank account example, we include redefinitions for the methods of *Account* in the subclasses *BounusAccount* and *SavingAccount*; the original redefinition of the method *credit* in *BonusAccount* is preserved. As an example of these trivial redefinitions, in the class *BonusAccount* we define

$$\mathbf{meth}\ getBalance\ \widehat{=}\ \mathbf{super}.getBalance$$

This is an abbreviation for

$$\mathbf{meth}\ getBalance\ \widehat{=}\ (\mathbf{res}\ bal : \mathbf{double} \bullet \mathbf{super}.getBalance(bal))$$

since, as already mentioned, all methods are parametrised commands.

4.4 Eliminate super

As mentioned in the previous step, before moving methods up, we need to make sure that their bodies do not contain references to **super**, otherwise the program semantics may not be preserved. This is because, when moving up a method that includes a method call of the form $\mathbf{super}.m$, instead of referring to a method m of the immediate superclass C, we may end up referring to a method m of a superclass of C. Furthermore, when we move such a method to **object**, the resulting program is invalid, since **super** cannot appear in **object**.

Our approach to eliminate **super** relies on Law 28, which is a form of copy rule for calls of the form $\mathbf{super}.m$ in a class C, based on a declaration of m in the immediate superclass of C. Since in the previous step we introduced a definition for all methods available in a class, a method called via **super** is always declared in the immediate superclass of the class where the call appears. Therefore, we can exhaustively apply Law 28 to eliminate all method calls using **super**.

This elimination process starts at the immediate subclasses of **object** and moves downwards. As the methods of **object** cannot refer to **super**, and all attributes are already public at this point, the condition of Law 28 is valid for the immediate subclasses of **object**. After eliminating **super** from those classes, the condition will be valid for their immediate subclasses, and so on.

```
class object
  pub number : string; balance : double; acc : Account
  pub prox : AccountList; bonus : double
end

class Account
  meth credit ≙ (val value : double • self.balance := self.balance + value)
  meth getBalance ≙ (res bal : double • bal := self.balance)
  meth getNumber ≙ (res n : string • n := self.number)
  meth setNumber ≙ (val n : string • self.number := n)
end

class BonusAccount extends Account
  meth credit ≙ (val value : double •
    self.balance := self.balance + value; self.bonus := value ∗ 0.05)
  meth getBonus ≙ (res bon : double • bon := self.bonus)
  meth getBalance ≙ (res bal : double • bal := self.balance)
  meth getNumber ≙ (res n : string • n := self.number)
  meth setNumber ≙ (val n : string • self.number := n)
end

class SavingAccount extends Account
  meth interest ≙
    var bal : double • self.getBalance(bal); self.credit(bal ∗ 0.1)

  meth credit ≙ (val value : double • self.balance := self.balance + value)
  meth getBalance ≙ (res bal : double • bal := self.balance)
  meth getNumber ≙ (res n : string • n := self.number)
  meth setNumber ≙ (val n : string • self.number := n)
end
...
• ...
```

Fig. 6. Bank account program—eliminate **super**

For our example, the result of the previous and of this step is partially shown in Figure 6. The main command is not affected. All classes explicitly define all methods that are available for their objects directly, or rather, without using calls to the corresponding methods of the superclass. We use the fact that $(pds \bullet (pds \bullet c)(\alpha pds))$ is equivalent to $(pds \bullet c)$, in any context; this is convenient for our use of the abbreviated notation **meth** $m \ \widehat{=}$ **super**.m.

Observe that the elimination of the original occurrence of **super** (in the method *credit* of *BonusAccount*) is justified by exactly the same law as the elimination of the occurrences introduced by the previous step.

4.5 Trivial Cast Introduction

In general, both to change the type of an object identifier (when this is an attribute) and to move method declarations up in the class hierarchy require that

the occurrences of these identifiers are cast. For example, consider the assignment $ac := \textbf{self}.acc$ in the body of the method *lookup* (see Figure 4). Both *ac* and *acc* are declared with type *Account*. Changing the type of *acc* to **object** clearly makes this assignment invalid; nevertheless, if *acc* is cast to *Account* the assignment is still valid. Further, occurrences of **self** (as in the assignment $x := \textbf{self}$) changes their type when they are moved up to a superclass, which might result in ill-typed assignments; therefore, they must be cast. Occurrences of **super** have been eliminated in the previous step.

Law 32 is sufficient to introduce trivial casts to non-assignable expressions in any program, including the main command. Figure 7 presents part of the bank account program that results from including the necessary casts. In the main command, the global variables, which have a primitive type, are not cast. Also, the existing cast (in class *AccountLit*) is not touched. As a result, all non-assignable expressions are cast, either because they were in the original program, or because casts were introduced by the current step of our reduction strategy.

4.6 Move Methods Up

After having eliminated **super** and introduced casts, we can safely move methods up to **object**. This is justified by Laws 23 and 24. We apply the first one when the method declaration that we want to move up is a redefinition of a method declared in the immediate superclass. The second should be applied when the method that we want to move is not a redefinition. We start applying Laws 23 and 24 from the bottom of the class hierarchy and move upwards towards **object**. The application of Law 23 introduces new occurrences of **self** in the program. These need to be cast as described in Section 4.5.

Using this strategy, the conditions for applying Law 23 are always valid: at this stage, all attributes are public and declared in **object**, and the method bodies do not use the **super** construct. This also explains why most of the conditions for applying Law 24 from left to right are valid. The only proviso we need to worry about are those related to the declaration of m in B and in its superclasses and subclasses. Since now every class redefines the methods in its superclass, if m is declared in C, but not in B, then it is not declared in any superclass of B. It is also not declared in any subclass of B, as, similarly to attribute names (see Section 4.2), we assume that method names are only reused for redefinitions.

For our example, all the methods of *AccountList* go directly to **object**. Figure 8 presents the result of this step. The method *credit* of *BonusAccount*, and that of *SavingAccount* are combined with the method *credit* of *Account*. One method is moved up first, and then the other one, in any order; the result is a method definition that tests for all the possible dynamic types of an *Account* object; this method declaration is moved up to **object**. Similarly, the other redefined methods are combined and moved all the way up to **object**. The program in Figure 8 can be simplified if we consider that an alternation of the form **if** $b \rightarrow c \,[]\, \neg\, b \rightarrow c$ **fi** can be simplified to c, as this is the

```
class object
  pub number : string; balance : double; acc : Account
  pub prox : AccountList; bonus : double
end

classAccount
  meth credit ≙ (val value : double •
    ((Account)self).balance := ((Account)self).balance + value)
  meth getBalance ≙ (res bal : double • bal := ((Account)self).balance)
  meth getNumber ≙ (res n : string • n := ((Account)self).number)
  meth setNumber ≙ (val n : string • ((Account)self).number := n)
end

class BonusAccount extends Account
  meth credit ≙ (val value : double •
    ((BonusAccount)self).balance := ((BonusAccount)self).balance + value;
    ((BonusAccount)self).bonus := value * 0.05)
  ...
end

class SavingAccount extends Account
  meth credit ≙ (val value : double •
    ((SavingAccount)self).balance := ((SavingAccount)self).balance + value)
  ...
end

class AccountList
  meth lookup ≙ (val number : string; res ac : Account •
    if (((AccountList)self).acc = null) → ac := null
    [] ¬ (((AccountList)self).acc = null) →
        var n : string • ((Account)((AccountList)self).acc).getNumber(n);
          if (n = number) → ac := ((AccountList)self).acc
          [] ¬ (n = number) →
              ((AccountList)((AccountList)self).prox).lookup(number, ac)
          fi
    fi)
end

• var acc : Account •
    acc := (Account)new SavingAccount;
    ((Account)acc).setNumber("21.342 − 7");
    ((Account)acc).credit(200);
    if (((Account)acc) is SavingAccount) → ((SavingAccount)acc).interest()
    [] ¬ (((Account)acc) is SavingAccount) → skip
    fi
```

Fig. 7. Bank account program—trivial cast introduction

command to be executed regardless of the condition b (see Law 39; other laws of alternation can also be applied to combine and simplify nested alternations. Nevertheless, this is not relevant for the purpose of obtaining a program in our normal form.

Exercise 9. Complete the program in Figure 8 with all the methods moved to class **object**.

Exercise 10. Identify the necessary laws of conditionals and of type test (**is**) that are necessary to simplify the bodies of the methods moved to class **object** as a result of the previous exercise. As an example, the body of the method *credit* in Figure 8 can be simplified to

meth *credit* $\widehat{=}$ (**val** *value* : **double** •
  if $\neg$ (**self is** *BonusAccount*) $\rightarrow$ **self**.*balance* := **self**.*balance* + *value*
  $[\!]$ (**self is** *BonusAccount*) $\rightarrow$
    self.*balance* := **self**.*balance* + *value*; **self**.*bonus* := *value* ∗ *0.05*
  fi

4.7 Change Type to object

Here we use the laws that formalise the fact that the types of attributes, variables, and parameters can be replaced with a supertype, if all non-assignable occurrences of these identifiers in expressions are cast: Laws 21, 25, 26, and 31. The exhaustive application of these laws, instantiating the type T' with **object**, allows the replacement of the types of all class-type identifiers with the **object** class. The provisos of the laws are valid, since we already have casts in expressions, and every class is a subclass of **object**. Variables of primitive types, including global variables, which we assume to be of a primitive type, are not affected by this reduction step. Figure 9 presents the effect of changing types to **object** in our example program.

4.8 Cast Elimination

After the previous step, the trivial casts introduced previously are not trivial anymore, since the types of the identifiers were changed to **object**. Furthermore, the program may include arbitrary casts previously introduced by a developer. Therefore, the laws we use to eliminate casts are different from those we use to introduce them.

Since a type cast may occur arbitrarily nested in an expression, it is convenient to reduce expressions to a simple form, so that we can consider only a fixed number of patterns. This form is as defined in the BNF for expressions (see Section 2), with arbitrary expressions (denoted by e) replaced with variables. The reduction of an arbitrary expression to this form is a reduction strategy in itself. Nevertheless, it is a very standard one, and is not presented here; this kind of reduction strategy can be found in [228].

```
class object
  pub number : string; balance : double; acc : Account
  pub prox : AccountList; bonus : double

  meth lookup ≙ (val number : string; res ac : Account •
    if (((AccountList)self).acc = null) → ac := null
    [] ¬ (((AccountList)self).acc = null) →
        var n : string •
          ((Account)((AccountList)self).acc).getNumber(n);
          if (n = number) → ac := ((AccountList)self).acc
          [] ¬ (n = number) →
                ((AccountList)((AccountList)self).prox).lookup(number, ac)
          fi
    fi)

  meth credit ≙ (val value : double •
    if ¬ (((Account)self) is SavingAccount) →
        if ¬ (((Account)self) is BonusAccount) →
            ((Account)self).balance := ((Account)self).balance + value)
        [] (((Account)self) is BonusAccount) →
            ((BonusAccount)self).balance := ((BonusAccount)self).balance + value;
            ((BonusAccount)self).bonus := value * 0.05)
        fi
    [] (((Account)self) is SavingAccount) →
        ((SavingAccount)self).balance := ((SavingAccount)self).balance + value)
    fi)
  ...
end

class Account end

class BonusAccount extends Account end

class SavingAccount extends Account end

class AccountList end

• var acc : Account •
    acc := (Account)new SavingAccount;
    ((Account)acc).setNumber("21.342 − 7");
    ((Account)acc).credit(200);
    if (((Account)acc) is SavingAccount) → ((SavingAccount)acc).interest()
    [] ¬ (((Account)acc) is SavingAccount) → skip
    fi
```

Fig. 8. Bank account program—methods in **object**

To deal with the elimination of casts in the remaining expression patterns, we use Laws 33 and 30, and others that are similar and omitted here. At this stage of our reduction strategy, all casts can be eliminated. The static role of each cast is trivially fulfilled as a consequence of the fact that the type of each object identifier is **object**, and that all methods and attributes have been moved to the **object** class. Therefore, the provisos of each law are always satisfied. As a result, the exhaustive application of these laws eliminates all casts in the program, as illustrated in Figure 9.

4.9 Method Elimination

The purpose of this step is to eliminate all method calls and then all method declarations, keeping in the **object** class only attribute declarations. For method call elimination, we need only Law 29, which can be regarded as a version of the copy rule. The reason is that we deal with dynamic binding when we move methods up to the **object** class. In fact, there are no method redefinitions at this point, since all methods are in **object**.

In this step, we apply Law 29 exhaustively. Before doing so, however, we need to change all recursive calls of the form $le.m$. We eliminate them by defining the method m with the use of the recursive command $\mu\, X \bullet c$, in such a way that recursive calls become references to X. The law that can be used to perform this change is standard and omitted.

After all calls to a method are replaced with its body using Law 29, the method definition itself can be eliminated using Law 27. These two laws are sufficient to eliminate all methods. There is no particular order to be followed; methods can be eliminated in any order. Even in the case where a method m invokes a method n, it is possible to eliminate m first, since in every place where m is invoked, we can replace this invocation by the body which includes an invocation to n; this is no problem since n is still in scope. At this point there are no private attributes, method redefinitions, or references to **super**.

As illustration, in Figure 10 we present part of the main command that shows the elimination of the calls $acc.setNumber(\texttt{"}21.342 - 7\texttt{"})$ and $acc.credit(200)$.

Exercise 11. Complement the main program in Figure 10 and simplify it using the laws identified in the previous exercise.

4.10 Summary of the Strategy

An arbitrary program can be reduced to *subtype normal form* by simply applying the steps described in Sections 4.1–4.9, in that order, eventually renaming attributes and methods to avoid naming conflicts. Although our normal form reduction strategy provides reassurance as to the expressiveness of our set of laws, it might be surprising that some of the laws presented in Section 3 are not referenced here. This is a consequence of the fact that our subtype normal form preserves classes, attributes, type tests, and object creation. We decided to aim at this normal form because it is close to an imperative program

```
class object
  pub number : string; balance : double;  acc : object
  pub prox : object;    bonus : double
  meth lookup ≙ (val number : string; res ac : object •
    {self is AccountList};
    if (self.acc = null) → ac := null
    [] ¬ (self.acc = null) →
        var n : string • {self is AccountList}; {self.accis Account};
          self.acc.getNumber(n);
          if (n = number) → {self is AccountList}; ac := self.acc
          [] ¬ (n = number) →
              {self is AccountList}; {self.prox is AccountList};
              self.prox.lookup(number, ac)
          fi
    fi)
  meth credit ≙ (val value : double •
    {self is Account};
    if ¬ (self is SavingAccount) →
        {self is Account};
        if ¬ (self is BonusAccount) →
            {self is Account}; {self is Account};
            self.balance := self.balance + value
        [] (self is BonusAccount) →
            {self is BonusAccount}; {self is BonusAccount};
            self.balance := self.balance + value;
            {self is BonusAccount}; self.bonus := value ∗ 0.05
        fi
    [] (self is SavingAccount) →
        {self is SavingAccount}; {self is SavingAccount};
        self.balance := self.balance + value
    fi)
  ...
end

classAccount end
class BonusAccount extends Account end
class SavingAccount extends Account end
class AccountList end

• var acc : object •
    {newSavingAccountis Account}; acc := new SavingAccount;
    {accis Account}; acc.setNumber("21.342 − 7");
    {accis Account}; acc.credit(200);
    {accis Account};
    if (accis SavingAccount) → {acc is SavingAccount}; acc.interest()
    [] ¬ (accis SavingAccount) → skip
    fi
```

Fig. 9. Bank account program—types changed to **object**, and elimination of casts

```
class object
  pub number : string; balance : double; acc : object
  pub prox : object; bonus : double
end
classAccount end
class BonusAccount extends Account end
class SavingAccount extends Account end
class AccountList end

• var acc : object •
    {new SavingAccount is Account}; acc := new SavingAccount;
    {acc is Account}; acc.number :="21.342 − 7";
    {acc is Account}; acc.credit(200);
    {acc is Account};
    if ¬ (acc is SavingAccount) →
        {acc is Account};
        if ¬ (acc is BonusAccount) →
            {acc is Account}; {acc is Account}; acc.balance := acc.balance + value
        [] (acc is BonusAccount) →
            {acc is BonusAccount}; {acc is BonusAccount};
            acc.balance := acc.balance + value;
            {acc is BonusAccount}; acc.bonus := value ∗ 0.05
        fi
    [] (acc is SavingAccount) →
        {acc is SavingAccount}; {acc is SavingAccount};
        acc.balance := acc.balance + value
    fi
    ...
```

Fig. 10. Bank account program in subtype normal form

and its reduction process is entirely algebraic; as mentioned before, reduction to a pure imperative form requires some sort of encoding of the object data model.

Exercise 12. The step for introducing trivial casts (Section 4.5) can be carried out before that for introducing trivial occurrences of **super** (Section 4.3). Analyse why changing the order of these steps would not affect the reduction strategy. Verify whether any other permutation of the steps would be possible.

5 Formal Refactoring

One of the main applications of the laws introduced in Section 3 is the formal derivation of refactorings. In fact, developers often wish to refactor programs or to define new refactorings, but are usually not so sure about the necessary preconditions and code transformations. Our laws give them a basis for proving that the transformations they apply or define preserve behaviour and, therefore,

are indeed refactorings. In this section, we first illustrate how a simple program can be refactored by systematically applying our laws, and then we present and derive, using the previously introduced laws, one refactoring as an equivalence law (a more extensive list can be found in [73]).

5.1 Refactoring a Banking Application

We consider a banking application where a user interface class accesses system services through a facade, which, in turn, accesses a collection of accounts. This sort of design is typical of well-structured layered object-oriented applications.

class *GUI*
pri f : *Facade*
pri *fields* : *GUIFields*
meth *action* $\widehat{=}$
  var a : *Account* $\bullet$
    $a :=$ **new***Account*(); ...
    self.f.*add*(a)
end

class *Facade*
  pri c : *Collection*
  meth *add* $\widehat{=}$ (**val** a : *Account* $\bullet$
    self.*log*("*Adding* ..."));
    var e : **bool** $\bullet$ ...
      if $\neg e \rightarrow$ **self**.c.*add*(a) **fi**)
  meth *log* $\widehat{=}$ (**val** s : **string** $\bullet$ ...)
end

For simplicity, some details are omitted. For example, constructors do not appear; we assume that they initialise the attributes by creating objects. The *action* method in *GUI* sets the fields of a with information from *fields* and then invokes a facade service, *add*, which adds an account to the collection as long as it contains no account with the same number.

The class *Facade* also keeps a log of the transactions on bank accounts; this is the main concern of the method *log*. We could further improve this design by using the wrapper pattern [96] in order to extract the calls to *log*. In this case, a wrapper would be responsible for logging and the facade would focus on business code only. We can actually show that this would be a valid refactoring of the original specification, just introduced, by applying, in a stepwise way, our laws.

We start the refactoring process by considering any context *CDS* containing at least classes *Collection* (with a method *add*), *Account* and *GUIFields* (both with the necessary *get* methods), and not redefining *Facade*'s method *add*. This will simplify the derivation, but is not strictly necessary, as we discuss later. We also consider any command c using only the methods and public attributes in this context and in the classes *GUI* and *Facade* just introduced.

By applying Law 27 from right to left, we introduce a new auxiliary method add', a fresh name, to *Facade*. By applying Law 17 from right to left we introduce, also with a fresh name, our wrapper, a direct subclass of **object**. We obtain the following equivalent, according to *CDS*, c, set of class declarations.

```
class GUI
  pri f : Facade
  pri fields : GUIFields
  meth action ≙
    var a : Account •
      a := new Account(); ...
      self.f.add(a)
end
class LWrapper
  pri f : Facade
  meth add ≙ (val a : Account •
    self.f.log(‘Adding ...’);
    self.f.add′(a))
end
```

```
class Facade
  pri c : Collection
  meth add ≙ (val a : Account •
    self.log("Adding ...");
    var e : bool • ...
      if ¬e → self.c.add(a) fi)
  meth add′ ≙ (val a : Account •
    var e : bool • ...
      if ¬e → self.c.add(a) fi)
  meth log ≙ (val s : string • ...)
end
```

The wrapper specification was chosen in such a way that calls to *Facade*'s *add* can actually be replaced by calls to the wrapper's *add* method.

Now Law 29 allows us to replace part of *add*'s body by a call to *add′*. This is possible for the following reasons: as *add′* is fresh, it is not redefined in *CDS*; **object** has no methods, so there are no **super** method calls in *add′*; from Laws 36 and 37, we can infer that {**self** ≠ **null** ∧ **self** ≠ **error**} is equivalent to **skip**, which can be freely introduced before any command; and part of *add*'s body can be easily represented by a parametrised command applied to *a*. However, notice that Law 19 should be applied before (from left to right) and after (from right to left) Law 29 to make the attributes public and then hide them again.

```
class GUI
  pri f : Facade
  pri fields : GUIFields
  meth action ≙•
    var a : Account •
      a := new Account(); ...
      self.f.add(a)
end
class LWrapper
  pri f : Facade
  meth add ≙ (val a : Account •
    self.f.log("Adding ...");
    self.f.add′(a))
end
```

```
class Facade
  pri c : Collection
  meth add ≙ (val a : Account •
    self.log("Adding ...");
    self.add′(a))
  methadd′ ≙ (val a : Account •
    var e : bool • ...
      if ¬e → self.c.add(a) fi)
  meth log ≙ (val s : string • ...)
end
```

As we want a specification of *GUI* that uses an object of *LWrapper* instead of an object of *Facade*, we apply Law 44 to obtain a simulation of the previous *GUI* class having a concrete *LWrapper* attribute *w*, a fresh new name, instead of the abstract attribute *f*. The coupling invariant *CI* is **self**.f = **self**.$w.f$, and the context *CDS′* for the simulation consists of *CDS* plus the previous *LWrapper*

and *Facade* classes. In the method bodies in GUI, simulation changes only the method call $\mathbf{self}.f.add(a)$, since the other statements do not access the attribute f; they just declare a variable a, initialise it, and set its fields. So the resulting simulation of GUI is as follows.

$$
\begin{array}{l}
\mathbf{class}\, GUI \\
\quad \mathbf{pri}\ w : LWrapper \\
\quad \mathbf{pri}\ fields : GUIFields \\
\quad \mathbf{meth}\ action \mathrel{\widehat{=}} \\
\qquad \mathbf{var}\ a : Account \bullet a := \mathbf{new} Account(); \ \ldots \mathbf{self}.w.f.add(a) \\
\mathbf{end}
\end{array}
$$

The application of the simulation law gives us $GUIf \preceq_{CDS',CI} GUIw$, where $GUIf$ and $GUIw$ denote the declarations of GUI with the f and w attributes. As discussed in Section 3.5, this implies the class refinement $GUIf \preccurlyeq_{CDS'} GUIw$, and then $GUIf \sqsubseteq_{CDS',c'} GUIw$ for any c' that uses only methods and public attributes in CDS' and $GUIf$. This is precisely what we need to infer:

$$GUIf\ LWrapper\ Facade \sqsubseteq_{CDS,c} GUIw\ LWrapper\ Facade$$

assuming that *LWrapper* and *Facade* denote here the respective class declarations; we just moved class declarations from the subscript to the sets of related class declarations.

This means that we can proceed with our derivation, still considering the context CDS, c, but now having only refinement instead of equivalence. Equivalence could be achieved by applying a simulation law in the opposite direction, but we omit the details here. By applying Law 29 from left to right, we can start to replace calls to *Facade*'s *add* by calls to *LWrapper*'s *add*: $\mathbf{self}.w.f.add(a)$ is equivalent to

$$
\begin{array}{l}
\{\mathbf{self}.w.f \neq \mathbf{null} \wedge \mathbf{self}.w.f \neq \mathbf{error}\}; \\
\mathbf{self}.w.f.log(\texttt{"}Adding \ldots\texttt{"});\ \mathbf{self}.w.f.add'(a)
\end{array}
$$

But, by the definition of assumption and the precondition weakening law, and using Law 38, we can infer that this is refined by

$$
\begin{array}{l}
\{\mathbf{self}.w \neq \mathbf{null} \wedge \mathbf{self}.w \neq \mathbf{error}\}; \\
\mathbf{self}.w.f.log(\texttt{"}Adding \ldots\texttt{"});\ \mathbf{self}.w.f.add'(a)
\end{array}
$$

which, by Law 29 from right to left, is equivalent to $\mathbf{self}.w.add(a)$. As before, this derivation is only possible by applying Law 19 before and after Law 29 to make the attributes public and then hide them again. Moreover, notice the following: *LWrapper* was introduced during the derivation, so it has no subclasses in CDS and, therefore, its methods are not redefined; we assumed that *Facade*'s *add* is not redefined in CDS.

This refinement leads to the following declarations.

```
class GUI
  pri w : LWrapper
  pri fields : GUIFields
  meth action ≙
    var a : Account •
      a := newAccount(); ...
      self.w.add(a)
end
class LWrapper
  pri f : Facade
  meth add ≙ (val a : Account •
    self.f.log("Adding..."); self.f.add′(a))
end
```

```
class Facade
  pri c : Collection
  meth add ≙ (val a : Account •
    self.log("Adding...");
    self.add′(a))
  meth add′ ≙ (val a : Account •
    var e : bool • ...
      if ¬e → self.c.add(a) fi)
  meth log ≙ (val s : string • ...)
end
```

Now the *GUI* call to *Facade*'s *add* was removed; other calls to *add* in the classes of *CDS* can be removed in a similar way. This changes the context we have been considering so far, but, as $cds =_{CDS,c} cds'$ is just an abbreviation to $cds\ CDS \bullet c = cds'\ CDS \bullet c$, we can still benefit from transitivity and proceed with our derivation. So, using Law 27, we eliminate the *add* method from *Facade*, and then, using Law 27 in the opposite direction, introduce a new *add* identical to *add′*. As *add* and *add′* now have the same body, we then replace calls to **self**.$f.add'(a)$ by **self**.$f.add(a)$, following an strategy similar to the one used to replace calls to *Facade*'s *add* by calls to *LWrapper*'s *add*. We can then eliminate *add′* and finally arrive at the refactored program below, which uses the wrapper pattern.

```
class GUI
  pri w : LWrapper
  pri fields : GUIFields
  meth action ≙
    var a : Account •
      a := newAccount(); ...
      self.w.add(a)
end
class LWrapper
  pri f : Facade
  meth add ≙ (val a : Account •
    self.f.log("Adding...");
    self.f.add(a))
end
```

```
class Facade
  pri c : Collection
  meth add ≙ (val a : Account •
    var e : bool • ...
      if e → self.c.add(a) fi)
  meth log ≙ (val s : string • ...)
end
```

Method and constructors bodies in *GUI* are the same as before except that now they access *w* instead of *f*: assignments to *f* were replaced by corresponding assignments to *w*, and *f* := **new** *Facade* became *w* := **new** *LWrapper*.

Method bodies in *Facade* are unchanged except for the commands extracted to *LWrapper*. The method *log* in *Facade* could be further moved to *LWrapper*, assuming it does not access c, but we omit the details here.

This refactoring process can be generalised to deal with arbitrary classes in the same situation as *GUI* and *LWrapper*. In other words, we could formalise a new refactoring law, as illustrated in the next section. We would then be able to perform the same transformations with other classes, having several methods and attributes, and also to extract command blocks, not just a single method call. Also, we could drop the restriction that *Facade*'s *add* is not redefined; this could be done by applying, during the refactoring, the normal form strategy used to eliminate dynamic binding, and later performing the reverse process.

5.2 Refactoring Laws

Refactoring laws denote a pair of refactorings, corresponding to applications of the law in each direction. They explicitly present generic transformations, and the conditions that must be satisfied in order to apply a refactoring. If the conditions are satisfied, the application of a refactoring to a program yields a new program that preserves the behaviour of the original one. Here we present the refactorings ⟨Pull Up Method⟩ and ⟨Push Down Method⟩, which combine and organise redundant method declarations. They are captured by the following law.

Refactoring 1 ⟨Pull Up/Push Down Method⟩

class A **extends** D
  adsa
  opsa
end class B **extends** A
  adsb
  meth $m \mathrel{\widehat{=}} (pds \bullet b)$
  opsb
end class C **extends** A
  adsc
  meth $m \mathrel{\widehat{=}} (pds \bullet b)$
  opsc
end
	$=_{cds,c}$	**class** A **extends** D
		  adsa
		  meth $m \mathrel{\widehat{=}} (pds \bullet b)$
		  opsa
		end class B **extends** A
		  adsb
		  opsb
		end class C **extends** A
		  adsc
		  opsc
		end

provided

(↔) (1) **super** and private attributes do not appear in b; (2) **super**.m does not appear in *opsb* or *opsc*; (3) m is not declared in a superclass of A in *cds*;

(→) m is not declared in *opsa*, and can only be declared in a class N, for any $N \leq A$, if it has parameters *pds*;

(←) (1) m is not declared in *opsb* or *opsc*; (2) $N.m$, for any $N \leq A$ such that $N \not\leq B$ or $N \not\leq C$, does not appear in *cds*, c, *opsa*, *opsb* or *opsc*. □

Applying this law from left to right corresponds to the first refactoring; the reverse direction corresponds to the other one. The class A that appears on the left hand-side of this law is the superclass of B and C, which declare a method m defined with the same parameters and body. As they have a common superclass and the method m is the same in both classes, we can move this method to the superclass. This helps maintenance as any modification will occur in just one method definition.

The provisos are similar to those of Laws 24 and 23. Notice that if the method in B uses elements of B through **self**, this method could not be the same as that of C, which clearly does not have access to B elements. We also require that m is not defined in a superclass of A, as otherwise the method m available in A ends up being different when we apply this refactoring.

Proof. In order to derive the above refactoring, we assume that the provisos are valid and begin the derivation with the class declarations on the left-hand side.

We cast occurrences of **self** in b to A, so that later we can move the methods m to A. Every command in which there is an occurrence of **self** is preceded by **skip**, the specification statement : [**true**, **true**]. By the definition of assumptions, we can write this specification statement as {**true**}. By Law 34, we have that the expression **self is** A is true in classes B and C. Applying this law, from right to left, we obtain the assumption {**self is** A}. In this way, every command with occurrences of **self** is now preceded by the assumption {**self is** A}. By applying Law 33, from right to left, we cast every occurrence of **self** in b with A. The result is denoted by b'.

By using Law 24, we move the method m that is declared in class B to its superclass A, obtaining the following declarations.

class A **extends** D
  $adsa$
  meth $m \mathrel{\widehat{=}} pds \bullet b'$
  $opsa$
end

class B **extends** A
  $adsb$
  $opsb$
end

class C **extends** A
  $adsc$
  meth $m \mathrel{\widehat{=}} pds \bullet b'$
  $opsc$
end

The next step moves the method m declared in C to its superclass A. However, this method is already declared in A. So, we use Law 23, which allows us to move a redefined method from a subclass to its superclass. This introduces an alternation in the method declared in the superclass, yielding the following result.

class A **extends** D
  $adsa$
  meth $m \mathrel{\widehat{=}} (pds \bullet$
    if $\neg$(**self is** C) $\rightarrow$ b'
    $[\!]$ **self is** $C \rightarrow b'$
    fi)
  $opsa$
end

class B **extends** A
  $adsb$
  $opsb$
end

class C **extends** A
  $adsc$
  $opsc$
end

The disjunction of the guards of the alternation we have introduced in the previous step is true, and the same command b' is guarded in both branches of the alternation. This allows us to apply Law 39 that reduces this alternation just to the command b'. Now we can remove the casts to A by applying Law 32, from right to left, obtaining the original command b. With this step we finish the proof of the refactoring $\langle$Pull Up/Push Down Method$\rangle$. □

Exercise 13. Using the laws discussed here, derive the Extract Method refactoring, which is considered the Rubicon of refactoring tools. Consider that you have imperative laws for transforming a command block into a parametrised command application. *This exercise is much harder than the previous ones.*

6 Conclusions

In this tutorial, we have explored the algebraic properties of some common object-oriented constructs available in languages like Java. Perhaps the major benefit of an algebraic presentation is that it encourages separation of concerns: the properties of each feature can be addressed in isolation. As an example, the elimination of method invocation (Law 29) has been dissociated from dynamic binding (Law 23), as well as from the behaviour of **super** (Laws 28 and 22). We have addressed completeness of the laws based on a normal form that mimics an imperative program: we presented a strategy to reduce an arbitrary program to such a form.

Unfortunately, the proposed laws, when viewed as transformation rules, are not compositional as the laws of pure imperative programming. Our laws, especially those involving classes, impose side conditions on the entire context (represented by the class declarations *cds* and the main program c). However, this seems inevitable when dealing with object-orientation, which is a paradigm heavily based on contextual information. It is worth stressing, nevertheless, that although the proposed laws rely on conditions on *cds* and c, these contextual elements are not modified by any of the laws.

Complementarily to exploring the properties of each language operator, we have addressed data refinement of classes, generalising Morgan's approach (that deals with single modules with private information) to class hierarchies possibly involving protected attributes. Together with the laws of commands and classes, this law serves as a powerful tool to prove more elaborate transformations, like refactorings, for instance, which are addressed in the previous section.

In the remainder of this section we discuss some additional issues: soundness of laws; the impact of a reference semantics, as opposed to a copy semantics; laws of other programming paradigms, particularly concurrency; and automation.

6.1 Soundness

A common criticism to the algebraic style is that merely postulating algebraic laws can give rise to complex and unexpected interactions between programming

constructions. This can be avoided by linking the algebraic semantics with a mathematical model in which the laws can be verified. Our laws have been proved sound [72] with respect to a weakest precondition semantics [48] defined by induction on the typing rules of the language. Besides the semantics itself, the cited work also introduces a notion of refinement, and defines equivalence and refinement of programs in a standard way.

The laws clearly need to be proved again if a different semantic model is adopted. The impact of adopting a reference semantics is discussed next.

6.2 Copy and Reference Semantics

As discussed in Section 2, we have adopted a copy semantics for our language; a major objective was to simplify the definition of a formal semantics (and a notion of refinement) [48] for an object-oriented language with features that are common in languages like Java. Nevertheless, it is interesting to examine, more closely, the algebraic laws that rely on copy semantics.

First we consider reference semantics assuming that expressions are side-effect free, as in our language. If a reference model is adopted, it is essential to define what is allowed as left expressions in assignments and the meaning of assignment. In languages like C and C++, which allow direct pointer manipulation, laws such as the one below for combining assignments are not valid.

$$(le := e_1;\ le := e_2) = (le := e_2[e_1/le])$$

Assuming that x is a pointer to an integer variable, and that y is an integer variable, in C and C++ the assignment $x := \&y$ assigns to x the address of y. As a result of this assignment, an *aliasing* is created between $*x$ (the value of the address stored in x) and y. Therefore, the sequence of assignments

$$*x := 1;\ *x := y + 1$$

has the effect of assigning 2 to y. On the other hand, the combined assignment $*x := y + 1$ increments the value of y by 1.

Some other languages like Java do not allow pointer manipulation, and attach a copy semantics to assignments involving variables of primitive types, say x and y. Therefore the effect of $x := y$ is confined to copying the value of y to x. As a result, if we adopt a Java-like reference semantics model for our language, the law for combining assignments would still be valid.

On the other hand, combining assignments to the same object-valued left-expression does not always preserve behaviour. For example, when a and b reference the same object, say an account of balance 2, the sequence of assignments

$$a.balance := b.balance + 1;\ a.balance := b.balance + a.balance$$

increases the balance attribute to 6, whereas the combined assignment

$$a.balance := b.balance + (b.balance + 1)$$

updates it to 5. Nevertheless, it is interesting to observe that the law still holds for assignments that do not involve access to attributes, even when the left-hand side of the assignments are object identifiers, like a and b above. For instance, the equality

$$(a := b;\ a := c) = (a := c)$$

holds because there is no operator that would allow the object expression c to generate a side-effect on b. Thus the only effect of both programs is to assign the object resulting from evaluating c to a.

This suggests that, if we adopt a Java-like reference semantics model for our language, then we can replace the law for combining assignments with two laws. The one below is similar to the original law, and is valid in general if x is an identifier not involving attribute access.

$$(x := e_1;\ x := e_2) = (x := e_2[e_1/x])$$

Another law deals with attribute access, with an explicit condition to preclude sharing (aliasing) between the object identifier being assigned and any subexpression of the second expression assigned to this identifier.

$$(le.a := e_1;\ le.a := e_2) = (le.a := e_2[e_1/le.a])$$
$$\textbf{\textit{provided}}\ \mathit{noSharing}(le, e_2)$$

The side condition is clearly not syntactic, and in general may require elaborate techniques, such as program analysis, to be discharged; a possible implementation of *noSharing* is out of the scope of this chapter.

In a similar way, the law that allows the distribution of assignments over a conditional also needs revising when considering a reference semantics. We need an extra condition concerning the absence of sharing between the expression in the assignment and those in the branches of the conditional.

$$(le := e;\ \textbf{if}\ [\!]i \bullet \psi_i \rightarrow c_i\ \textbf{fi}) = (\textbf{if}\ [\!]i \bullet \psi_i[e/le] \rightarrow (le := e;\ c_i)\ \textbf{fi})$$
$$\textbf{\textit{provided}}\ \mathit{noSharing}(le, \psi_i)$$

Surprisingly, perhaps, no other law of commands (including those introduced in Section 3.4 as well as the ones presented in [72]) are affected by a model of reference semantics as that of Java, adopted to our language.

The law for change of data representation, which generalises the traditional data refinement law for a single module [192] to class hierarchies, on the other hand, is also affected by reference semantics. To be valid in general, such a law would have to consider pointer confinement issues as, for example, in [20].

Side-effect in expressions is another aspect that may impact the validity of some laws; but this is a concern orthogonal to the semantic model (copy or reference semantics). In the presence of side-effects like, for instance, method calls as expressions, as allowed in Java, none of the laws considered above are valid, regardless whether we adopt a copy or a reference model.

In summary, reference semantics, as in Java, and side-effect free expressions (for instance, avoiding method calls in the expression language) cause a relatively small impact on the set of laws of our language, with copy semantics. Furthermore, in the absence of sharing, all laws of our language are valid in the context of a reference semantics as well.

Of course, these claims need to be formally proved. In [130], concepts of the unifying theories of programming (UTP) are used to formalise a semantic model based on that of Java for a language similar to ours. This model could be an interesting basis for the formal verification of both the revised laws discussed in this section and the ones that we claim are not affected by a reference semantics.

6.3 Laws of Other Programming Paradigms

This work shows that the laws of object-oriented programming naturally extend well-established laws of pure imperative programming [116]. This is not surprising given that, due to the concept of states in objects, object orientation is considered by some authors to be a variation of the imperative paradigm, rather than a strictly new paradigm.

As an orthogonal aspect, concurrency has been integrated to several programming paradigms; Java itself is an object-oriented and concurrent language. Nevertheless, laws that simultaneously address object-oriented and concurrent features seem to be a recent topic of research. The language *OhCircus* [52] extends *Circus* [255] with object-oriented features similar to those of our language. *Circus* itself combines CSP, Z, and Morgan's specification statements. Refinement laws for concurrent processes in *Circus* are presented in [230, 51]. At the moment, we are proposing laws for *OhCircus*, based on the ones presented here and those of *Circus*. A formal UTP semantics for *OhCircus* is being defined.

The modeling language UML-RT (UML for Real Time) also combines object-oriented and concurrent features. A formal semantics and laws for UML-RT are presented in [220, 229].

6.4 Automation

Besides contributing to the formal verification of behaviour preserving transformations, the results presented here are useful as a basis for the implementation of refactoring tools. The preconditions of our laws are purely syntactic, leading to refactorings that have syntactic preconditions that can be automatically discharged by tools. Moreover, our laws suggest essential refactorings, which should be provided by tools that allow the user to compose existing refactorings to define new ones [100]. Similarly, executable languages for specifying new refactorings from scratch [46] should be able to express the laws presented here.

Using CSP

Jim Davies

Oxford University Computing Laboratory
Wolfson Building, Parks Road
Oxford OX1 3QD UK

This is a tutorial on *Communicating Sequential Processes (CSP)*: a language for modelling patterns of behaviour. It explores the design of the language, and shows how it may be used to construct descriptions of behavioural properties and distributed systems. It explains also how the use of the language may be supported by verification tools.

In the next section, we motivate the use of communicating sequential processes to model and reason about complex systems. After that, we present the basic constructs of CSP that can be used to define sequential processes; a number of laws and examples provide the intuition. The semantic models of CSP are the subject of Section 3, where we also introduce the notion of refinement for processes. Sections 4 and 5 present more elaborate CSP constructs to compose processes, including those that model parallel behaviour; again, an extensive number of laws and examples are presented. Data aspects of a system can also be modelled in CSP; this is discussed in Section 6. An important application of CSP is communication protocols; their modelling is discussed in Section 7. Finally, in Section 8, we briefly present the main CSP tools: FDR and Probe.

1 Introduction

1.1 Complex Systems

The computing systems that we build are becoming closer to life; they are ever more: *pervasive*, delivering increased functionality in a wider variety of situations; *connected*, providing services and information on-demand; and *powerful*, holding, processing, and integrating larger amounts of data. As a result, they are rapidly becoming more *complex*.

The human ability to deal unaided with complexity is not increasing at the same rate. We need to augment our faculties with appropriate languages, methods, and tools; the question of what is appropriate is still being answered. It is not enough to look to older, established engineering disciplines for approaches and solutions.

The development of computing systems presents particular challenges in terms of: *scale*—the systems that we design need to be understood at many different layers of abstraction, and viewed from many different perspectives; *change*—the requirements upon these systems will change during development, as a result of changes in the environment, or changes in our understanding; *behaviour*—

A. Cavalcanti, A. Sampaio, and J. Woodcock (Eds.): PSSE 2004, LNCS 3167, pp. 64–122, 2006.

understanding structure and state is not enough, we need to understand also how the system will behave.

To understand complex systems, we must construct comprehensible, abstract models, including in each model only that information which is relevant to our immediate purpose. A model might focus upon the structure, or architecture, of a system; the data that it contains, and the effect of each operation upon that data; or the behaviour that it might exhibit. In this chapter, we will see how to construct comprehensible, abstract models of *behaviour*.

It is useful for us to distinguish between two kinds of behavioural model: *instance models* and *process models*. An instance model presents a particular scenario, giving an example of how a component may behave. The sequence diagram of Figure 1 illustrates the following sequence of interactions between a person *User* and a vending machine *Machine*: the user inserts a *50* cent coin, and then selects *orange*; in response, the machine will dispense a can of *orange* drink.

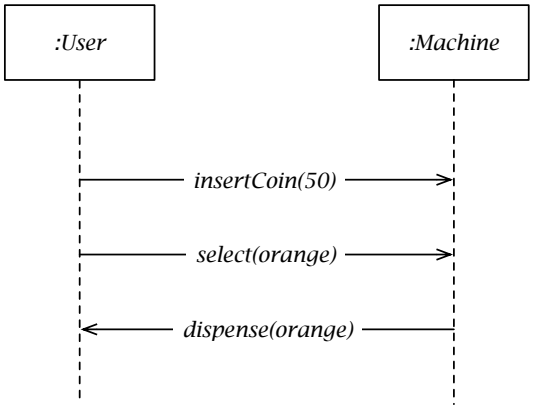


Fig. 1. A sequence diagram

A process model, on the other hand, presents a complete picture, describing all of the possible behaviours of a component with respect to a particular set of actions or events. The state diagram of Figure 2 describes all of the possible behaviours of a vending machine, while partitioning the state space of the component into two regions: *Ready* and *Select*.

In *Ready*, the insertion of a coin will result in a transition to *Select* if and only if the *value* of the coin, when added to the current *holding*, equals or exceeds the *price* of a can of drink. In *Ready*, any selection is ignored. In *Select*, a user selection results in the corresponding can of drink being dispensed; if the new value of *holding* is less than *price*, the component returns to the *Ready* state.

This diagram describes all of the possible behaviours of the machine with regard to the *insertCoin*, *select*, and *dispense* actions. It is an idealized representation, in that it does not take account of the fact that the machine may fill up with coins, or run out of cans; neither does it take account of the nondeter-

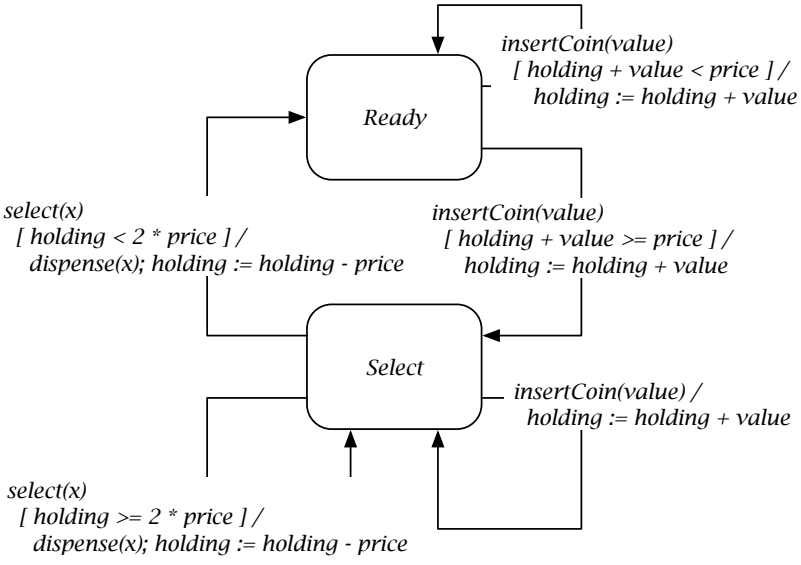


Fig. 2. A state diagram

minism that should result from the exclusion of this information (the capacity of the coinbox, the number of cans remaining) from the model.

Instance models are ideal for the description of scenarios, particularly those associated with requirements or tests. For a complete understanding, however, process models are necessary: they can tell us what the system *could* do—in any situation—as opposed what it *should* do—in a limited range of situations.

To be comprehensible, any complex model of behaviour must be presented in terms of sequential descriptions: it is difficult for a human being to follow two or more threads of a story at the same time. In general, a model of the behaviour of a complex system will involve more than one sequential description: for otherwise the multiplication of entities and states would make the narrative too long, and the choices too numerous to contemplate.

A model of a complex system will need to be presented as a composition of sequential descriptions, or model components, each explaining part of the overall pattern of behaviour. Furthermore, the properties of the model must be determined entirely by the properties of these sequential descriptions: our language of models must be *compositional*.

1.2 Abstract Events

The language introduced in this chapter is compositional for a wide range of behavioural properties: those expressible in terms of the occurrence and availability of *abstract events*. We will see how it can be used to compose process models of complex systems from simple sequential descriptions, and how it can be used to compare two models—instance or process—of the same system.

We will describe the behaviour of a component in terms of abstract events, points at which there is some relevant interaction between the component and its environment. This will be enough to explain how the component will behave in combination; it should also be all that we need to understand about the resulting system.

Abstract events are atomic and have no measurable duration. We may compare them to lines in planar geometry: they act as a boundary between behaviour before, and behaviour afterwards. They are also synchronous: if an event represents an interaction between two components, then it occurs for both components at the same time, or not at all.

These events are perfectly symmetric, having the same interpretation in each component: in particular, there is no notion of the source, or target, of an event. There is no limit upon the number of components involved: events represent abstract *transactions*, not point-to-point communications.

This notion of *event* is the ideal primitive for the abstract description of behaviour. Where there is a need to distinguish between different stages of an interaction, we may use an event for each distinction that we need to make. Where no such need exists, a single event will suffice.

In constructing a model, we will aim for a minimal set of events: just enough to express the properties that we wish to consider. We may wish to add further events—for clarity, or regularity, or to suggest features of an implementation—but we should remember that each new event increases the complexity of the model, and makes it harder to reason about.

Example 1 (Doors). The number of events needed to represent the actions of opening and closing a door will depend upon the purpose of the model in question. If the model is intended to help us understand the part of an alarm system, a single event may suffice: we may need only to model the action of opening the door. In reasoning about the behaviour of a lift control system, we may need several different events: to model the actions of the doors starting to open, reaching fully open, starting to close, and reaching fully closed.

1.3 Communicating Sequential Processes

Communicating Sequential Processes (CSP) is a language for the description and analysis of complex systems. In CSP: all models are process models; a complex model may be constructed from simple, sequential components; and behaviour is described in terms of abstract events.

The language of CSP has a precise, mathematical interpretation: the meaning of a model is the set of all behaviours that it admits; each behaviour is described in terms of sequences of events performed, and sets of events subsequently refused. This *semantics* supports a powerful theory of refinement and proof.

A number of tools have been developed to allow us to reason about complex models written in CSP. Interaction with these tools involves not only the process language, but also its semantics: individual behaviours are used to describe

particular scenarios. Because of this, the semantic language of behaviours is as important as the process language itself.

In the subsequent sections, we will introduce first the language of sequential models, then its semantics, and then the mechanisms for parallel composition and abstraction. As we do so, we will see how the language, the semantics, and the tools can be used together in reasoning about the behaviour of complex systems.

2 Processes

The language of CSP is an algebraic notation, with event names, process names, and operator symbols. In this section, we introduce the operators used to compose sequential descriptions. We also introduce a number of algebraic properties: each time we observe that two expressions are simply different ways of writing the same process, we learn something more about the meaning of our language.

2.1 Basic Processes and Events

The language of CSP has two basic processes: *Stop* and *Skip*; neither of these processes will ever engage in an event. The first represents the end of a pattern of behaviour: no more events will be performed. The second represents successful termination: again, no events will be performed, but if there is another component to describe subsequent behaviour, then this is the point at which that behaviour can begin.

If a is an event, and P is a process, then the expression $a \rightarrow P$ denotes a process that is ready to engage in a; should this event occur, the process will then behave as P. We call '$\rightarrow$' the *event prefix* operator: event a must happen first. We say 'ready to engage', because no component can simply 'do' an event: each event is a synchronisation with the environment; it can be performed only when every other component involved is ready.

Example 2 (Vending machines). The process

$$coffee \rightarrow Stop$$

is ready to engage in *coffee*; after this event, it will never do anything again. The process

$$coin \rightarrow (coffee \rightarrow Stop)$$

is ready to engage in *coin*. If this happens, the process is then ready to engage in *coffee*. Once both events have been performed, the process stops.

We may introduce a new process name with a simple equation: on the left is the new name; on the right is an expression in the process language.

$$Name = \ldots$$

Example 3 (Vending machines). We may use the process name *OCM* for our coffee machine:

$$OCM = coin \rightarrow (coffee \rightarrow Stop)$$

These definitions may be recursive, in that the process name being introduced may appear within the expression on the right-hand side.

$$Name = \ldots Name \ldots$$

Example 4 (Vending machines). A simple coffee machine accepts a coin, offers a coffee, and then returns to its initial state. We may describe this behaviour using a process CM, defined as follows:

$$CM = coin \rightarrow (coffee \rightarrow CM)$$

As far as the model is concerned, this component will accept coins, and offer coffee, indefinitely.

2.2 Choice

Our process language has two forms of choice: internal and external. The first of these represents nondeterminism, or uncertainty, regarding the behaviour of a component—it is as if the choice is made within the component. The second represents a menu of options for interaction—it is as if the choice is offered to the environment.

If P and Q are processes, then we write $P \sqcap Q$ to denote the internal choice between them. This describes a component that may behave as either P or Q. Whatever the factors are that influence or determine which of the two process descriptions will apply, they are not considered in this model.

Example 5 (Vending machines). The process *NVM* describes the behaviour of a vending machine that will decide, internally, whether to offer coffee or tea.

$$\begin{aligned} NVM = coin \rightarrow \\ (\; & coffee \rightarrow NVM \\ \sqcap \\ & tea \rightarrow NVM \;) \end{aligned}$$

An internal decision might be taken at any time before the result becomes apparent. It may be that the internal design of the component would only ever admit one of the two alternatives; or, at the other extreme, it may be that some hidden action or computation has taken place immediately before the interaction occurs. Our process language has the following property:

Law 47 (→–distribution). *For any event a, and any processes P and Q,*

$$a \rightarrow (P \sqcap Q) \; = \; (a \rightarrow P) \sqcap (a \rightarrow Q)$$

An event followed by an internal choice is the same as an internal choice between two processes, each starting with the event in question.

Example 6 (Vending machines). In the case of the nondeterministic vending machine, the internal choice between *coffee* and *tea* may be made before or after the coin is inserted:

$$\begin{array}{lcl} coin \rightarrow & = & (coin \rightarrow \\ \quad (\, coffee \rightarrow NVM & & \quad\; coffee \rightarrow NVM) \\ \quad\; \sqcap & & \sqcap \\ \quad\; tea \rightarrow NVM\,) & & (coin \rightarrow \\ & & \quad\; tea \rightarrow NVM) \end{array}$$

If P and Q are processes, we write $P \,\Box\, Q$ to denote the external choice between them. This describes a component that is offering to behave as either process: the environment can choose between them by agreeing to perform an event that is possible for exactly one of the two alternatives.

Example 7 (Vending machines). The following process describes the behaviour of a more conventional vending machine:

$$\begin{array}{l} VM = coin \rightarrow \\ \qquad (\, coffee \rightarrow VM \\ \qquad\; \Box \\ \qquad\; tea \rightarrow VM\,) \end{array}$$

This machine will respond to the insertion of a coin by offering the user a choice between *coffee* and *tea*.

An external choice creates a menu of processes. The presence of *Stop* in such a menu makes no difference to the overall behaviour: it is an option that cannot be chosen.

Law 48 ($\Box$–*Stop*). *For any process P,*

$$Stop \,\Box\, P \;=\; P$$

If an event is possible for two or more of the alternatives in an external choice, then the environment cannot use that event to decide between them. If it is performed, then the outcome is nondeterministic.

Law 49. *For any event a, and any processes P and Q,*

$$(a \rightarrow P) \,\Box\, (a \rightarrow Q) \;=\; (a \rightarrow P) \sqcap (a \rightarrow Q)$$

This is a special case of the *step law* for external choice, presented later.

If one of the options in a menu is an internal choice, then the resulting behaviour is the same as that of an internal choice of menus.

Law 50 ($\Box$–distribution). *For any processes P, Q, and R,*

$$P \,\Box\, (Q \sqcap R) \;=\; (P \,\Box\, Q) \sqcap (P \,\Box\, R)$$

If E is a set of events, and for each element e of E we can define a parameterized process $P(e)$, then the process

$$\Box\, e : E \bullet e \rightarrow P(e)$$

describes the behaviour of a component that offers the environment a choice of events from E. If event e is chosen, then the subsequent behaviour is that of the corresponding process $P(e)$.

This combination of event prefix and external choice is called *prefix choice*. It is a particularly important construction in our language of processes: not only does it allow us to describe a range of alternative actions as a single choice, but it is also an ideal basis for explaining the meaning of the other operators. If the process P satisfies

$$P = \Box\, e : A \bullet e \rightarrow P(e)$$

then we know which events P is willing to engage in *at the next step*, and what the consequence of each event would be. If we have this information for each component, then we can infer the same information for any composition of processes (although there may be an internal choice of menus to consider). The equation that expresses this inference is called the *step law* for the composition.

Step Law ($\Box$). If processes P and Q are such that

$$P = \Box\, e : A \bullet e \rightarrow P(e)$$

$$Q = \Box\, e : B \bullet e \rightarrow Q(e)$$

then

$$\begin{aligned} P \,\Box\, Q \;=\; \Box\, e : A \cup B \bullet e \rightarrow \\ &\text{if } e \in A \setminus B \text{ then} \\ &\quad P(e) \\ &\text{else if } e \in B \setminus A \text{ then} \\ &\quad Q(e) \\ &\text{else} \\ &\quad P(e) \sqcap Q(e) \end{aligned}$$

The composition $P \,\Box\, Q$ is ready to perform any event from the set $A \cup B$. If the event e is chosen, then what we know about the subsequent behaviour depends upon whether e is in:

- the difference $A \setminus B$, in which case it could only have been performed by P, and the subsequent behaviour must be described by $P(e)$;
- the difference $B \setminus A$, in which case it could only have been performed by Q, and the subsequent behaviour must be described by $Q(e)$;
- the intersection $A \cap B$, in which case it could have been performed by either process, and that behaviour can only be described as $P(e) \sqcap Q(e)$.

The conditional expression operator 'if B then ... else ...' saves us the trouble of having to present two equations, one for each of the possible values of the boolean expression B. It may be used for expressions of any type, but we will find it most useful for processes.

Another useful choice construct is *asymmetric choice*. If P and Q are processes, then we write $P \triangleright Q$ to denote the behaviour of a component that *may* offer P, but must offer Q. That is,

$$P \triangleright Q \;=\; (P \,\Box\, Q) \sqcap Q$$

There is either a menu choice between P and Q, or there is just Q. Taking into account the Laws 48 and 50, we could write the same process as $(P \sqcap STOP) \,\Box\, Q$, because

$$\begin{aligned} &(P \sqcap STOP) \,\Box\, Q \\ &\quad = (P \,\Box\, Q) \sqcap (STOP \,\Box\, Q) && \Box\text{-distribution} \\ &\quad = (P \,\Box\, Q) \sqcap Q && \Box\text{-comm (below)}, \Box\text{-}Stop \end{aligned}$$

The two choice operators admit the expected algebraic properties: for any processes P, Q, and R

- *idempotence*:
  $P \sqcap P = P$ ($\sqcap$-idem)
  $P \,\Box\, P = P$ ($\Box$-idem)
- *commutativity*:
  $P \sqcap Q = Q \sqcap P$ ($\sqcap$-comm)
  $P \,\Box\, Q = Q \,\Box\, P$ ($\Box$-comm)
- *associativity*:
  $P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap R$ ($\sqcap$-assoc)
  $P \,\Box\, (Q \,\Box\, R) = (P \,\Box\, Q) \,\Box\, R$ ($\Box$-assoc)

They also satisfy another distributive property presented below.

Law 51. *If P and Q are processes, then*

$$(P \,\Box\, Q) \sqcap R = (P \sqcap R) \,\Box\, (Q \sqcap R)$$

That is, an internal choice can be distributed over a menu—an external choice of processes.

In an internal choice with R, the menu $P \,\Box\, Q$ behaves as another menu, with two options corresponding to P and Q, and internal choices to indicate that, whichever option we would prefer, we may still be faced with R instead. This law serves as a reminder that, in CSP, things change (and choices are resolved) only when events occur.

2.3 Sequencing

If P and Q are processes, we write $P \,;\, Q$ to describe the behaviour of a component that behaves according to the sequential composition of P and Q: that is, it behaves as P until that process terminates successfully, and then behaves as Q. You may recall that we use the basic process *Skip* to represent a point at which a process may terminate successfully.

In CSP, we use shared events to indicate points at which one process may affect the behaviour of another. The two processes in a sequential composition can never share an event: indeed, one does not even start until the other has terminated. As a result, there is a side-condition to the step law for this operator, restricting the scope of the declaration of the event variable e.

Step Law (;). If the process P is such that

$$P = \Box\, e : A \bullet e \rightarrow P(e)$$

and Q is any process, then

$$P \,;\, Q \;=\; \Box\, e : A \bullet e \rightarrow (P(e) \,;\, Q)$$

provided that e is not a parameter to the definition of Q (that would suggest that e had some meaning, other than that of 'some event from menu A').

This law tells us that, whenever the behaviour of the first process can be described as a prefix choice, the second process makes no immediate contribution to the behaviour of a sequential composition.

We regard an empty prefix choice as equivalent to *Stop*: no event can be chosen, and no progress can be made.

Step Law (Stop). If P is any process, then

$$\Box\, e : \varnothing \bullet e \rightarrow P \;=\; \mathit{Stop}$$

In combination with the step law for sequential composition, this tells us that

$$\mathit{Stop} \,;\, P \;=\; \mathit{Stop}$$

The only way that the second process can start is if the first process reaches a point where *Skip* appears.

Law 52 (*Skip*–;). *If P is any process, then*

$$\mathit{Skip} \,;\, P = P$$

Successful termination results in a seamless transition to the next process: no event is observed.

Example 8 (Breakfast). If the processes *Juice*, *Cereal*, and *Coffee* are defined by

$$Juice \;=\; glass \rightarrow juice \rightarrow Skip$$

$$Cereal \;=\; bowl \rightarrow cereal \rightarrow milk \rightarrow Skip$$

$$Coffee \;=\; cup \rightarrow coffee \rightarrow sugar \rightarrow Skip$$

then we may use their sequential composition to describe the pattern of behaviour associated with a (rather sequential) breakfast:

$$Breakfast \;=\; Juice \,;\, Cereal \,;\, Coffee$$

Coffee does not begin until *Cereal* has terminated; *Cereal* itself does not begin until *Juice* has terminated.

As termination is not marked by any external event, the *Breakfast* process could have been defined using prefix and (eventually) successful termination.

$$\begin{aligned} Breakfast \;=\; & glass \rightarrow juice \rightarrow \\ & bowl \rightarrow cereal \rightarrow milk \rightarrow \\ & cup \rightarrow coffee \rightarrow sugar \rightarrow Skip \end{aligned}$$

Sequential composition distributes through $\sqcap$ in both arguments.

Law 53 (;–distribution). *For any processes P, Q, and R,*

$$P \,;\, (Q \sqcap R) = (P \,;\, Q) \sqcap (P \,;\, R)$$
$$(P \sqcap Q) \,;\, R = (P \,;\, R) \sqcap (Q \,;\, R)$$

The fact that we cannot observe this abstract form of termination means that we are not able to prevent it from occurring; if *Skip* appears as one of the alternatives in a menu, then that menu behaves as an asymmetric choice.

Law 54. *For any process P,*

$$P \,\Box\, Skip \;=\; P \triangleright Skip$$

The initial events of P may be available; alternatively, the process may behave exactly as *Skip*.

A second form of sequential composition is described using the interrupt operator. If P and Q are processes, then we write $P \triangle Q$ to represent the behaviour of a component that behaves as P, except that an external choice with Q is presented at every stage. Should Q be chosen, then P is forgotten, making no further contribution to the overall pattern of behaviour.

Step Law ($\triangle$). If processes P and Q are such that

$$P = \Box\, e : A \bullet e \rightarrow P(e) \quad \text{and} \quad Q = \Box\, e : B \bullet e \rightarrow Q(e)$$

then

$$\begin{aligned} P \triangle Q = \Box\, e : A \cup B \bullet e \rightarrow & \text{ if } e \in A \setminus B \text{ then} \\ & \quad P(e) \triangle Q \\ & \text{else if } e \in B \setminus A \text{ then} \\ & \quad Q(e) \\ & \text{else} \\ & \quad (P(e) \triangle Q) \sqcap Q(e) \end{aligned}$$

provided that e does not appear in process expression Q.

The interrupt operator has the effect of a repeated external choice: as we might expect, it is associative; it is also distributive in both arguments.

Example 9 (Pinball). A game of pinball may be interrupted at any point by a *tilt* event, when the machine is pushed too hard. If the game-playing activity is described by the following process

$$Game = launch \rightarrow out \rightarrow launch \rightarrow out \rightarrow launch \rightarrow out \rightarrow Skip$$

then we may describe the behaviour of the pinball machine using interrupt and sequential composition:

$$Pinball = start \rightarrow (Game \triangle tilt \rightarrow Skip)\ ;\ Pinball$$

The resulting behaviour is illustrated by the state machine in Figure 3.

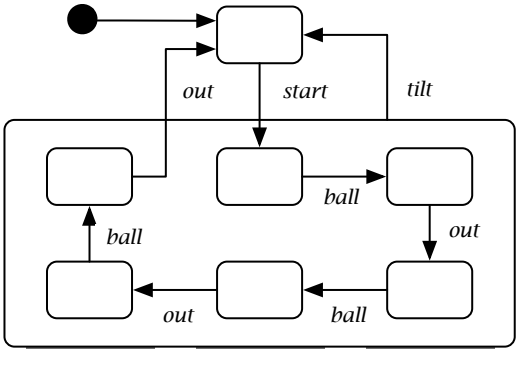


Fig. 3. Pinball states

3 Behaviours

Although we will write processes to represent our requirements and designs, the question of whether a process is suitable or unsuitable will often be answered in terms of individual behaviours. In this section, we see how individual behaviours can be represented using the simple mathematical concepts of finite sets and sequences, and how these concepts can be used to give a meaning, or semantics, to our process notation.

3.1 Traces

One way of describing a behaviour is to write down a list of the events that occur, in order of occurrence. Such a list, or sequence, of events is called a *trace*. The information content of a trace is similar to that of an instance model: for example, the sequence diagram of Figure 1 corresponds to the trace

$$\langle insertCoin.50, select.orange, dispense.orange \rangle$$

This trace is written using compound events, in which the second part of the event name corresponds to a value chosen from some datatype; we will introduce compound events and datatypes in Section 6.

Example 10 (Vending machines). The following trace could be performed by NVM:

$$\langle coin, coffee \rangle$$

It could also be performed by the more conventional VM.

Any process may be identified with a set of traces: the set of sequences of events that it could perform. If CSP denotes the set of process terms, and $Event$ denotes the set of all events, then we may define a function

$$traces : CSP \to \mathbb{P}(\operatorname{seq} Event)$$

mapping each term to a set of sequences of events. This function will help us to understand the meaning of the language; it is also important in the construction and *use* of analysis tools.

We define the function *traces* using recursion: the trace set of a compound process is defined in terms of the trace sets of its components. The first clauses in this recursive definition—for the basic processes, event prefix, choice, and sequential composition—are shown in Table 1.

We use the special symbol ✓ to distinguish between the behaviours of the two basic processes. *Stop* has just one trace: the empty trace $\langle\rangle$. *Skip* has two: it may have done nothing as yet, and so the trace is empty; or it may have terminated, in which case the trace contains the special symbol ✓.

For any event a, and any process P, any non-empty trace of the prefix process $a \to P$ must begin with the event a, and the remainder of the trace must be a

Table 1. The *traces* function

$$traces\ [\![Stop]\!] \ = \ \{\ \langle\rangle\ \}$$

$$traces\ [\![Skip]\!] \ = \ \{\langle\rangle, \langle\checkmark\rangle\}$$

$$traces\ [\![a \rightarrow P]\!] \ = \ \{\ \langle\rangle\ \} \cup \{\ tr_P : \ traces\ [\![P]\!] \bullet \langle a\rangle \frown tr_P\ \}$$

$$traces\ [\![P \ \Box\ Q]\!] = traces\ [\![P]\!] \cup traces\ [\![Q]\!]$$

$$traces\ [\![P \sqcap Q]\!] = traces\ [\![P]\!] \cup traces\ [\![Q]\!]$$

$$traces\ [\![P\ ;\ Q]\!] \ = \ \{\ tr_P : \ traces\ [\![P]\!];\ tr_Q : \ traces\ [\![Q]\!] \mid (\checkmark \notin \operatorname{ran} tr_P \wedge tr_Q = \langle\rangle) \vee tr_P \frown \langle\checkmark\rangle \in traces\ [\![P]\!] \bullet tr_P \frown tr_Q\ \}$$

possible trace of P. The trace set of this process is thus formed as a union: of the set containing the empty trace, and the set of all traces formed as a concatenation $\langle a\rangle \frown tr_P$, where tr_P is a trace of P.

Example 11 (Vending machines). Recall the definition of the one-shot coffee machine:

$$OCM = coin \rightarrow (coffee \rightarrow Stop)$$

There are three possible traces of this process:

$$traces\ [\![OCM]\!] = \{\ \langle\rangle, \langle coin\rangle, \langle coin, coffee\rangle\ \}$$

Any trace of a choice process must be a possible trace of at least one of the two alternatives. This applies whether the choice is external or internal:

Example 12 (Vending machines). The two one-shot vending machines

$$OVM = coin \rightarrow (\ coffee \rightarrow Stop \ \Box\ tea \rightarrow Stop\ )$$

$$NOVM = coin \rightarrow (\ coffee \rightarrow Stop \ \sqcap\ tea \rightarrow Stop\ )$$

both admit the same set of traces:

$$traces\ [\![OVM]\!] \ = \ traces\ [\![NOVM]\!] \ = \{\ \langle\rangle, \langle coin\rangle, \langle coin, coffee\rangle, \langle coin, tea\rangle\ \}$$

Any trace of a sequential composition will be formed as the concatenation of two traces: one from the first process, and another from the second. The second process can do nothing until the first terminates, and so the second trace must be empty unless the first trace could be followed by a $\checkmark$.

Example 13 (Breakfast). The processes *Juice* and *Cereal* admit the following sets of traces:

$$traces\,[\![Juice]\!] \;=\; \{\,\langle\rangle, \langle glass\rangle, \langle glass, juice\rangle, \langle glass, juice, \checkmark\rangle\,\}$$

$$traces\,[\![Cereal]\!] \;=\; \{\,\langle\rangle, \langle bowl\rangle, \langle bowl, cereal\rangle, \langle bowl, cereal, milk\rangle, \langle bowl, cereal, milk, \checkmark\rangle\,\}$$

The traces of the sequential composition *Juice* ; *Cereal* are given by:

$$\begin{aligned} traces\,[\![Juice\,;\,Cereal]\!] \;=\; \{&\langle\rangle, \langle glass\rangle, \langle glass, juice\rangle, \\ &\langle glass, juice, bowl\rangle, \\ &\langle glass, juice, bowl, cereal\rangle, \\ &\langle glass, juice, bowl, cereal, milk\rangle, \\ &\langle glass, juice, bowl, cereal, milk, \checkmark\rangle\,\} \end{aligned}$$

The successful termination of the first component does not mean termination of the entire process, so we do not include a ✓ at this point in the trace.

By considering traces, we can tell whether or not a process may accept a particular sequence of events, in the sense that this sequence could be performed if all of the other processes involved were ready to synchronize. However, we cannot tell whether or not the process might also reject a particular sequence.

Example 14 (Vending machines). The two one-shot vending machines introduced earlier have the same set of traces:

$$\begin{aligned} OVM = coin \rightarrow\ & (\ coffee \rightarrow Stop \\ & \Box \\ & tea \rightarrow Stop\ ) \end{aligned} \qquad \begin{aligned} NOVM = coin \rightarrow\ & (\ coffee \rightarrow Stop \\ & \sqcap \\ & tea \rightarrow Stop\ ) \end{aligned}$$

In particular, both of them may accept the sequence $\langle coin, coffee\rangle$. However, one of them, *NOVM*, may also reject it, by refusing to engage in *coffee* after *coin* has occurred; the internal choice at this point may be resolved in favour of the process $tea \rightarrow Stop$.

If we were concerned only with the behaviour of our processes in a sequential context, then it would be enough to record two sets of traces: those that may be accepted, and those that might also be rejected. If a possible trace could also be refused, then we will call it a 'refusal trace' of the process; we might define a corresponding function

$$\text{refusal traces} : \; CSP \rightarrow \mathbb{P}(\text{seq}\, Event)$$

to explain how the 'refusal traces' of a process are determined by the operators used to define it.

If we avoid any operator, such as internal choice ($\sqcap$), that may introduce nondeterminism into a process description, then we will find that the two sets of traces are disjoint. For any *deterministic* process P,

$$traces\,[\![P]\!] \cap \text{refusal traces}\,[\![P]\!] \;=\; \varnothing$$

For other processes, this intersection will be non-empty.

Example 15 (Vending machines).

$$\text{refusal traces}\,[\![OVM]\!] \;=\; \{\,\}$$

$$\text{refusal traces}\,[\![NOVM]\!] \;=\; \{\,\langle coin, coffee\rangle, \langle coin, tea\rangle\,\}$$

In a sequential (or 'single-threaded') context, where the environment may present the process a single sequence of events, this is all the information we need. Here, it does not matter that *NOVM* will guarantee to offer either coffee or tea; all that matters is that whichever sequence we offer, we might be disappointed.

In a concurrent context, where the environment may offer more than one event at the same time, 'refusal traces' information is not enough.

Example 16 (Vending machines). Consider the following process, describing the behaviour of a possibly-crooked vending machine:

$$\begin{aligned} COVM = coin \rightarrow (\; & coffee \rightarrow Stop \\ & \sqcap \\ & tea \rightarrow Stop \\ & \sqcap \\ & Stop\;) \end{aligned}$$

This has the same traces, and the same 'live traces', as *NOVM*. However, in an environment that is ready to perform both *coffee* and *tea* after *coin*, *NOVM* is guaranteed to make progress, whereas *COVM* might prove disappointing.

3.2 Failures and Divergences

If we wish to reason about the availability of events, based upon a nondeterministic description, in a concurrent context, then we need to consider more than just 'refusal traces'. We need to consider the various combinations of events that may be refused after each trace. We call such a combination a 'refusal set', or simply a *refusal*.

Example 17 (Vending machines). The process *COVM* admits the following refusals following the trace $\langle coin\rangle$:

$$\varnothing, \{coin\}, \{coffee\}, \{tea\}, \{coin, coffee\}, \{coin, tea\}, \{coffee, tea\},\\ \{coin, coffee, tea\}$$

Two of these sets are *not* refusals of *NOVM*:

$$\{coffee, tea\}, \{coin, coffee, tea\}$$

the process *NOVM* cannot refuse both *coffee* and *tea*: whichever way the internal choice is resolved, one of these events must be available.

Table 2. The *failures* function

$$failures\,[\![Stop]\!] = \{\, ref : \mathbb{P}\,Event \bullet (\langle\rangle, ref)\,\}$$

$$failures\,[\![Skip]\!] = \{\, tr : \operatorname{seq} Event;\ ref : \mathbb{P}\,Event \mid (tr = \langle\rangle \land \checkmark \notin ref) \lor tr = \langle\checkmark\rangle\,\}$$

$$failures\,[\![a \to P]\!] = \{\, tr : \operatorname{seq} Event;\ ref : \mathbb{P}\,Event \mid tr = \langle\rangle \land a \notin ref \lor head(tr) = a \land (tail(tr), ref) \in failures\,[\![P]\!]\,\}$$

$$failures\,[\![P \,\Box\, Q]\!] = \{\, tr : \operatorname{seq} Event;\ ref : \mathbb{P}\,Event \mid tr = \langle\rangle \land (tr, ref) \in failures\,[\![P]\!] \cap failures\,[\![Q]\!] \lor tr \neq \langle\rangle \land (tr, ref) \in failures\,[\![P]\!] \cup failures\,[\![Q]\!]\,\}$$

$$failures\,[\![P \sqcap Q]\!] = failures\,[\![P]\!] \cup failures\,[\![Q]\!]$$

$$failures\,[\![P\ ;\ Q]\!] = \{tr_P, tr_Q : \operatorname{seq} Event;\ ref : \mathbb{P}\,Event \mid \checkmark \notin \operatorname{ran} tr_P \land tr_Q = \langle\rangle \land (tr_P, ref \cup \{\checkmark\}) \in failures\,[\![P]\!] \lor tr_P \frown \langle\checkmark\rangle \in traces\,[\![P]\!] \land (tr_Q, ref) \in failures\,[\![Q]\!] \bullet (tr_P \frown tr_Q, ref)\}$$

We will call each possible (*trace*, *refusal*) pair a *failure* of the process. It corresponds to the result of a failed experiment, in which the process is offered all of the events in the refusal set after engaging in the trace, and refuses to accept any of them, making further progress impossible. We define a function *failures* that returns the set of all failures associated with a particular process term:

$$failures : CSP \to \mathbb{P}(\operatorname{seq} Event \times \mathbb{P}\,Event)$$

This gives (almost) the definitive meaning for our language of processes: see the equations of Table 2.

The basic process *Stop* can refuse any combination of events after the empty trace. *Skip*, on the other hand, can refuse anything after $\langle\rangle$ *except* the special event $\checkmark$. Once this event has been performed—that is, after the trace $\langle\checkmark\rangle$ has been recorded—*Skip* can refuse any combination of events *ref*.

The first event performed by the process $a \to P$ must be a: while the trace is empty, it may refuse any combination of events that does not include a. If a has been performed, the subsequent behaviour—the rest of the trace, and the refusal set—must be a possible behaviour of P.

A behaviour of a choice process could be a behaviour of either alternative. If the choice is external, then an additional constraint applies: when the trace is empty, a set of events can be refused only if both process would refuse that combination. (This is not *quite* the whole story: an external choice can also refuse any set not including $\checkmark$ when *Skip* is one of the alternatives.)

Example 18 (Vending machines). Failures information is enough to distinguish between the three different one-shot vending machines:

- $(\langle coin \rangle, \{coffee\})$ is a failure of $NOVM$, but not of OVM;
- $(\langle coin \rangle, \{coffee, tea\})$ is a failure of $COVM$, but not of $NOVM$ or OVM.

The failures semantics of a sequential composition is complicated by the fact that we do not record the ✓ event that marks the successful termination of the first component. We know that any trace must consist of a sequence (possibly empty) of events performed by the first process, followed by another sequence (again, possibly empty) performed by the second, but where one stops and the other starts may not be fully determined.

If the behaviour is entirely due to the first component, then clearly ✓ cannot appear in the trace. Furthermore, ✓ must not have happened, for otherwise the refusal set would be due to the second component. To exclude this possibility, we consider only those behaviours in which the first component could perform the trace, and then refuse ✓ (in addition to whatever else is being refused).

If both components have made a contribution to the behaviour, then the first must have terminated: whatever trace was performed could have been followed, in the traces of that process, by the special event ✓. In this case, only the second process makes a contribution to the refusal set.

Example 19 (Breakfast). The process *Juice* can perform the trace

$$\langle glass, juice, \checkmark \rangle$$

and the process *Cereal* has the failure

$$(\langle bowl \rangle, \{bowl, milk\})$$

The sequential composition *Juice* ; *Cereal* thus has a failure

$$(\langle glass, juice, bowl \rangle, \{bowl, milk\})$$

The equations in Table 2 give a precise meaning to some of the operators in our process language, describing the failures of basic processes, and showing how the failures of each kind of compound process may be derived from the failures of its components. They support the step laws of the previous section, and help us to understand why a particular failure might represent a possible behaviour of a given process.

We *could* use them to compute (an expression for) all of the failures of a given process but, in all but the most trivial of cases, the results would be incomprehensible. Instead, we will compute all of the given (control) states of a process, identify any problematic behaviours, and then report the results in terms of failures of process components: the role of the equations is to assist us in understanding and interpreting these results.

Whatever means of computation is adopted, whether we focus upon behaviours or control states, we must decide how to give a meaning to processes

defined using recursion. The equations above would be enough to explain the meaning of a process such as

$$a \rightarrow P \,\Box\, b \rightarrow P$$

in terms of the meaning of P, but what if we are in the process of giving a meaning to P itself? What if the above expression appears on the right-hand side of the equation used to define P?

$$P \;=\; a \rightarrow P \,\Box\, b \rightarrow P$$

In this case, the meaning of P is determined as a fixed point of a function: the function upon processes given by

$$F(X) \;=\; a \rightarrow X \,\Box\, b \rightarrow X$$

With this definition, $P = F(P)$: whatever process P is, it is not changed by the application of F.

It is a simple matter to assign a trace semantics to a process such as P. We consider the smallest set of traces required for a process to satisfy the equation; we can construct this through repeated application of the function, starting with the basic process *Stop*: that is, if $P = F(P)$, then

$$\mathit{traces}\,[\![P]\!] \;=\; \bigcup\{\, n : \mathbb{N} \bullet F^n(\mathit{Stop})\,\}$$

That is

$$\mathit{traces}\,[\![\mathit{Stop}]\!] \cup \mathit{traces}\,[\![F(\mathit{STOP})]\!] \cup \mathit{traces}\,[\![F(F(\mathit{Stop}))]\!] \cup \ldots$$

Each application of the function should reveal more about the traces of the process being defined.

Example 20 (Vending machines). The right-hand side of the equation defining *VM* can be seen as the result of applying a function F, where...

$$F(X) \;=\; \mathit{coin} \rightarrow (\mathit{coffee} \rightarrow X \,\Box\, \mathit{tea} \rightarrow X)$$

The semantic equations for $\Box$ and $\rightarrow$ us that

$$\begin{aligned}\mathit{traces}\,[\![F(X)]\!] = \{\langle\rangle, \langle \mathit{coin}\rangle\} \cup \{\, tr_X : \mathit{traces}\,[\![X]\!] \bullet \{\langle \mathit{coin}, \mathit{coffee}\rangle \frown tr_X\} \cup \\ \{\, tr_X : \mathit{traces}\,[\![X]\!] \bullet \{\langle \mathit{coin}, \mathit{tea}\rangle \frown tr_X\}\end{aligned}$$

and hence that

$$\mathit{traces}\,[\![F(\mathit{Stop})]\!] = \{\langle\rangle, \langle \mathit{coin}\rangle, \langle \mathit{coin}, \mathit{coffee}\rangle, \langle \mathit{coin}, \mathit{tea}\rangle\}$$

$$\begin{aligned}\mathit{traces}\,[\![F(F(\mathit{Stop}))]\!] = \{&\langle\rangle, \langle \mathit{coin}\rangle, \langle \mathit{coin}, \mathit{coffee}\rangle, \langle \mathit{coin}, \mathit{tea}\rangle, \\ &\langle \mathit{coin}, \mathit{coffee}, \mathit{coin}\rangle, \langle \mathit{coin}, \mathit{tea}, \mathit{coin}\rangle, \\ &\langle \mathit{coin}, \mathit{coffee}, \mathit{coin}, \mathit{coffee}\rangle, \langle \mathit{coin}, \mathit{coffee}, \mathit{coin}, \mathit{tea}\rangle, \\ &\langle \mathit{coin}, \mathit{tea}, \mathit{coin}, \mathit{coffee}\rangle, \langle \mathit{coin}, \mathit{tea}, \mathit{coin}, \mathit{tea}\rangle\}\end{aligned}$$

and so on. In this case, applying the function n times will yield all of the traces of P of length $\leq 2n$.

If every instance of a process name on the right-hand side of the equation is prefixed—or *guarded*—by at least one event, then the defining function has exactly one fixed point. If not, then there are other fixed points. For example, the equation

$$Q \;=\; (a \rightarrow Q) \,\Box\, Q$$

is satisfied by both QA and QB, where

$$QA \;=\; a \rightarrow QA$$

$$QB \;=\; (a \rightarrow QB) \,\Box\, (b \rightarrow QB)$$

which are two quite different processes.

If we are interested only in the traces of a process, then we may arrive at a suitable fixed point by repeated application of the defining function. However, using failures information alone, we cannot consistently assign a meaning to processes whose names appear unguarded within their defining equations: whatever approach we take, some of the algebraic laws stated in Section 2 are certain to fail.

The solution is to record, separately, the fact that we have reached a point at which the future behaviour is described by a process whose defining equation is unsatisfactory. We can do this by identifying each process with another set of traces, its *divergences*:

$$divergences : \; CSP \rightarrow \mathbb{P}(\mathrm{seq}\, Event)$$

A sequence of events d is a divergence of process P if the behaviour of P, after performing trace d, is partly described by a process name that appears unguarded in its defining equation.

Example 21 (Vending machines). The vending machine VM has no divergences:

$$divergences\, [\![VM]\!] \;=\; \varnothing$$

However, the diverging vending machine, described by

$$DVM \;=\; coin \rightarrow MENU$$

$$MENU \;=\; (coffee \rightarrow DVM \,\Box\, tea \rightarrow DVM \,\Box\, MENU)$$

has the trace $\langle coin \rangle$ as a divergence:

$$\langle coin \rangle \in divergences\, [\![DVM]\!]$$

we can say nothing about the behaviour of this process once the first event *coin* has been performed.

The above example illustrates a common misconception. In writing a menu for the right-hand side of an equation, we might be tempted to express the fact that 'if nothing external happens, we stay here' by adding the name of the current stage as an extra alternative. This is inappropriate: the equation is there to tell us which *observable* events may happen next; adding the current process name to the choice tells us nothing new, and renders the equation useless for the purpose of describing failures.

The notion of divergences lets us complete the failures semantics; we can assign a meaning to processes whose names appear unguarded within their defining equations. This allows us to ensure that any tools that we build to analyse our descriptions will produce correct, consistent results. The meaning is not terribly helpful in itself: after any trace that is also a divergence, a process may refuse *or perform* any combination of events: we can say nothing at all about the future behaviour—it is completely unconstrained, or *chaotic*.

When we start to reason about our process descriptions, however, we will find the notion of *Chaos* extremely useful. As part of a specification, it is a way of saying that, at this point, we do not care—anything is allowed. As part of a design, it is a way of saying that, at this point, we do not know—anything could happen.

3.3 Refinement

Our analysis of process descriptions will depend mostly upon the theory of process refinement. A process is a *refinement* if it introduces no new behaviours into the semantic set:

Definition 1 (Refinement). *If P and Q are processes, then we say that P is refined by Q, written*

$$P \sqsubseteq Q$$

if every behaviour of Q is also a behaviour of P.

Our two notions of behaviour—traces and failures (with divergences)—give rise to different, consistent, notions of refinement.

Definition 2 (Traces refinement). *If P and Q are processes, then we say that Q is a trace refinement of P, written*

$$P \sqsubseteq_T Q$$

if every trace of Q is also a trace of P.

We can use trace refinement to check whether particular sequences of interaction are possible for a component described by a process P. To do this, we write a process that is capable of these sequences of interaction, and no others; if this is a trace refinement of P, then every one of the sequences in question is indeed possible.

Example 22 (Vending machines). We may confirm that our vending machine might perform the sequence of interaction $\langle coin, coffee, coin, tea \rangle$, and every initial subsequence thereof, by defining

$$I = coin \rightarrow coffee \rightarrow coin \rightarrow tea \rightarrow Stop$$

and checking that $VM \sqsubseteq_T I$.

We can check also whether every sequence of interaction possible for a component described by P is acceptable according to some specification. To do this, we write a process that is capable of performing every acceptable sequence, and no others. If P is a trace refinement of this process, then every sequence of interaction will be acceptable.

Example 23 (Vending machines). We might wish to confirm that our coffee machine CM will never dispense a drink—either coffee or tea—without receiving at least one coin per drink in payment. We may do this by defining

$$S = coin \rightarrow Ready$$

$$Ready = coin \rightarrow Ready \,\Box\, coffee \rightarrow S \,\Box\, tea \rightarrow S$$

and checking that $S \sqsubseteq_T CM$. In doing this, we have decided that it would be acceptable for the machine to accept any number of coins before offering a drink, but that—whenever a drink is dispensed—at least one more coin must be inserted before a further drink is offered.

We can use trace refinement to demonstrate that a particular sequence of interaction may be possible for a given component. However, unless the component description is deterministic in this regard, we cannot demonstrate that a sequence *must* be possible: that is, at every step, the component will be able to perform the next event. To reason about availability, we need to consider failures and divergences.

Definition 3 (Failures–divergences refinement). *If P and Q are processes, then we say that Q is a failures–divergences refinement of P, written*

$$P \sqsubseteq_{FD} Q$$

if every failure of Q is also a failure of P, and every divergence of Q is also a divergence of P.

With failures and divergences, it is less useful to check that a component description is refined by a process describing a particular behaviour. The fact that a failure $(tr, \varnothing)$ is a possible behaviour tells us nothing about availability of the trace tr: for any prefix tr' of tr, and any non-empty set of events E, the failure (tr', E) might be another possible behaviour; tr could also be unavailable.

Instead, we write a specification that requires that the sequence is available (anything else is unacceptable behaviour) and check that our component description is a refinement of this specification.

Example 24 (Vending machines). If we wish to insist that the vending machine VM be capable of performing the following trace

$$\langle coin, coffee, coin, tea \rangle$$

offering each event in sequence, then we may define

$$S1 = coin \rightarrow S2 \,\Box\, (tea \rightarrow Chaos \sqcap coffee \rightarrow Chaos \sqcap Stop)$$

$$S2 = coffee \rightarrow S3 \,\Box\, (coin \rightarrow Chaos \sqcap tea \rightarrow Chaos \sqcap Stop)$$

$$S3 = coin \rightarrow S4 \,\Box\, (coffee \rightarrow Chaos \sqcap tea \rightarrow Chaos \sqcap Stop)$$

$$S4 = tea \rightarrow Chaos \,\Box\, (coin \rightarrow Chaos \sqcap coffee \rightarrow Chaos \sqcap Stop)$$

and check that $S1 \sqsubseteq_{FD} VM$. This requires that the four events of the trace are offered in sequence. Should the component and its environment choose to participate in some other event, we no longer care about subsequent behaviour; the same is true once the sequence is completed. We use the immediately-divergent process *Chaos* to indicate this; any process refines *Chaos*.

Example 25 (Vending machines). The nondeterministic machine NVM will not satisfy the requirement expressed using $S1$ above. It does not guarantee to offer *coffee* after the first coin; its failure semantics includes the pair

$$(\langle coin \rangle, \{coffee\})$$

This is not a failure of $S1$, and hence $S1 \not\sqsubseteq_{FD} NVM$.

If processes P and Q describe the same component, and Q is a failures–divergences refinement of P, then the two descriptions are consistent. Q is a better description or, equivalently, a description of a better design: everything that P guarantees, in terms of behaviour, is also guaranteed by Q.

Example 26 (Vending machines). The machine NVM guarantees to offer either tea or coffee once a coin has been inserted. On the trace $\langle coin \rangle$, it may refuse $\{tea\}$, or $\{coffee\}$, but not the combination $\{tea, coffee\}$. This process is refined by the vending machine VM, which guarantees to offer coffee, and also guarantees to offer tea: none of the sets above would be refused. It is also refined by the coffee machine CM (Of course, CM cannot offer tea, but NVM never guaranteed to do so).

Failures–divergences refinement is the definitive basis for the comparison of component descriptions. It corresponds to the notion of equality used in the algebraic identities and step laws, in that for any processes P and Q,

$$P \sqsubseteq_{FD} Q \wedge Q \sqsubseteq_{FD} P \;\Leftrightarrow\; P = Q$$

It matches the notion of nondeterminism introduced by the internal choice operator, in that for any P and Q

$$P \sqsubseteq_{FD} Q \;=\; P \Leftrightarrow P \sqcap Q$$

Finally, it is preserved by each of the operators used to construct processes (some of which we have yet to introduce). If P is refined by Q, then

$$a \rightarrow P \;\sqsubseteq\; a \rightarrow Q$$

$$P \,\Box\, R \;\sqsubseteq Q \,\Box\, R$$

$$P \sqcap R \;\sqsubseteq Q \sqcap R$$

$$P \parallel R \;\sqsubseteq Q \parallel R$$

$$P \setminus E \;\sqsubseteq Q \setminus E$$

Results established using refinement remain valid in any context.

4 Parallel Composition

We use the choice and sequencing operators to construct processes that describe sequential (components of) components, or properties that we understand. We may then compose these processes—using *parallel composition*—for the purposes of analysis and design.

4.1 Alphabets

In order to place processes in parallel, we must identify the set of events in which each is intended to participate. We cannot simply infer this from the events that appear in the description: some of these may turn out to be unavailable; an equivalent formulation might not mention them, but should lead to the same behaviour in the parallel composition.

If P and Q are processes, and αP and αQ are sets of events, then we may write

$$P \,[\![\, \alpha P \mid \alpha Q \,]\!]\, Q$$

to denote the parallel composition of P and Q, in which P and Q are intended to participate in every event from sets αP and αQ, respectively. In this parallel composition, we call αP the *alphabet* of P; P cannot perform any event in αP without the cooperation of the environment; the environment cannot perform any event in αP without the cooperation of P.

In the parallel composition $P \,[\![\, \alpha P \mid \alpha Q \,]\!]\, Q$, the second process Q forms part of the environment of P, and the cooperation of both processes is required if any event from the set $\alpha P \cap \alpha Q$ is to occur.

Step Law (‖). If processes P and Q are such that

$$P = \Box\, e : A \bullet e \rightarrow P(e)$$

$$Q = \Box\, e : B \bullet e \rightarrow Q(e)$$

then

$$\begin{aligned} P \,[\![\, \alpha P \mid \alpha Q \,]\!]\, Q \;=\; & \Box\, e : (A \setminus \alpha Q) \bullet e \rightarrow (P(e) \,[\![\, \alpha P \mid \alpha Q \,]\!]\, Q) \\ & \Box \\ & \Box\, e : (B \setminus \alpha P) \bullet e \rightarrow (P \,[\![\, \alpha P \mid \alpha Q \,]\!]\, Q(e)) \\ & \Box \\ & \Box\, e : (A \cap B) \bullet e \rightarrow (P(e) \,[\![\, \alpha P \mid \alpha Q \,]\!]\, Q(e)) \end{aligned}$$

Example 27 (Boxes). Two people are loading boxes onto a truck: a red box, a blue box, and a green box. Matthew is to load the red, while Marina loads the blue; the green is particularly heavy, and both of them will be needed to lift it. Matthew is happy to load the red and green boxes in either order:

$$Matthew \;=\; red \rightarrow green \rightarrow Stop \;\Box\; green \rightarrow red \rightarrow Stop$$

whereas Marina insists upon loading the green box before the blue:

$$Marina \;=\; green \rightarrow blue \rightarrow Stop$$

The parallel composition

$$Matthew \,[\![\, \{red, green\} \mid \{blue, green\} \,]\!]\, Marina$$

is equivalent to the following description:

$$\begin{array}{l} red \rightarrow green \rightarrow blue \rightarrow Stop \\ \Box \\ green \rightarrow (\; red \rightarrow blue \rightarrow Stop \\ \qquad\qquad \Box \\ \qquad\qquad blue \rightarrow red \rightarrow Stop \;) \end{array}$$

If a process is assigned the same alphabet wherever it appears, then we need define αP only once. If αP and αQ have been defined, then we may write $P \parallel Q$ in place of $P \,[\![\, \alpha P \mid \alpha Q \,]\!]\, Q$.

Example 28 (Boxes). If

$$\alpha Matthew = \{red, green\}$$

$$\alpha Marina = \{blue, green\}$$

then

$$Matthew \parallel Marina \;=\; Matthew \,[\![\, \{red, green\} \mid \{blue, green\} \,]\!]\, Marina$$

A parallel composition can terminate only when each of the processes involved is ready to do so. It is as if the special event ✓ is implicitly present in every process alphabet.

Law 55 ($\|$–*Skip*).

$$Skip \parallel Skip = Skip$$

Example 29 (Breakfast). We might imagine a parallel version of our breakfast process:

$$ParallelBreakfast = Juice \parallel Coffee \parallel Cereal$$

This process can terminate only when *Juice* has performed both *glass* and *juice*, *Cereal* has performed *bowl*, *cereal*, and *milk*, and *Coffee* has performed *cup*, *coffee*, and *sugar*.

4.2 Multi-way Synchronisation

The abstract events in CSP correspond to transactions, rather than point-to-point communications: there is no limit to the number of processes that may be involved. In particular, an event that is shared between two processes P and Q may also be shared with a third process R: in the combination $P \parallel Q \parallel R$, the intersection $\alpha P \cap \alpha Q \cap \alpha R$ may be non-empty.

Example 30 (Boxes). Alan drives the truck, and his cooperation is required for the loading of any of the boxes.

$$\alpha Alan = \{red, green, blue\}$$

For some reason, he insists that the *green* box is loaded before the *red*, and the *red* before the *blue*.

$$Alan = green \rightarrow red \rightarrow blue \rightarrow Stop$$

In the parallel combination *Matthew* $\parallel$ *Marina* $\parallel$ *Alan* all three processes are involved in the event *green*. Fortunately, the order of events insisted upon by *Alan* is allowed by the parallel combination of *Matthew* and *Marina*; all three boxes can be loaded into the truck.

To ensure that the parallel operator is associative, we must be consistent in the way that we assign alphabets to processes. If S is defined by

$$S = P \parallel Q$$

and we have assigned alphabets to P, Q, and S, then we must take care that

$$\alpha S = \alpha P \cup \alpha Q$$

Example 31 (Boxes). The parallel combination *Matthew* $\parallel$ *Marina* $\parallel$ *Alan* could also be written as

$$(Matthew \parallel Marina) \,[\![\, \alpha Matthew \cup \alpha Marina \mid \alpha Alan \,]\!]\, Alan$$

Furthermore, we must be consistent in the way that we assign alphabets to processes constructed using the choice and sequencing operators. If S is defined by any of the following equations,

$$S = a \rightarrow P \qquad S = P \sqcap Q$$

$$S = P \,\Box\, Q \qquad S = P \,;\, Q$$

and alphabets are assigned to S and P, then αS must be equal to αP (and similarly for Q). The set of events in which a particular process is intended to participate does not change as the process evolves. To avoid confusion, we require also that any event appearing in an event prefix, such as $a \rightarrow P$ above, is included in the alphabet of the resulting process. The use of *Stop* or *Skip* in a process definition does not constrain our choice of alphabets.

4.3 Concurrency and Nondeterminism

The parallel operator has the same distributive property as the other operators in our language. If one of the components is an internal choice, then the parallel composition could also be written as an internal choice of parallel compositions: one for each alternative.

Law 56 ($\parallel$–distribution). *For any processes P, Q, and R,*

$$P \parallel (Q \sqcap R) = (P \parallel Q) \sqcap (P \parallel R)$$

$$(P \sqcap Q) \parallel R = (P \parallel R) \sqcap (Q \parallel R)$$

We may use the distributive and step laws to rewrite a parallel composition of processes into sequential form. Before we do this, we may wish to give names to various stages of each of the component processes: if each named stage will be described as an external choice over a set of events, and the behaviour after each event is described as a (possibly single) nondeterministic choice of named stages, then it will be easy to keep track of the rewriting process.

Example 32 (Vending machines). An alternative design for a vending machine might be described as the parallel composition of two processes. The first of these represents a component that accepts a coin and determines whether or not a drink should be offered.

$$\begin{aligned} \alpha Coins &= \{coin, approve, return\} \\ Coins &= coin \rightarrow Approve \sqcap Return \\ Approve &= approve \rightarrow Coins \\ Return &= return \rightarrow Coins \end{aligned}$$

The event *approve* represents the successful validation of a coin; *return* represents the return of a coin that has not been successfully validated.

The second process represents a component that responds to *approve* by offering a choice of tea or coffee:

$$\alpha Drinks = \{approve, tea, coffee\}$$

$$Drinks = approve \rightarrow Ready$$

$$Ready = tea \rightarrow Drinks \;\Box\; coffee \rightarrow Drinks$$

As we intend to use the step law for $\|$ to construct a sequential view of a parallel combination of these components, we have taken care to write their description as a collection of named stages. The resulting descriptions are easily translated into state diagrams, in which each transition is labelled with the name of an event: see Figure 4.

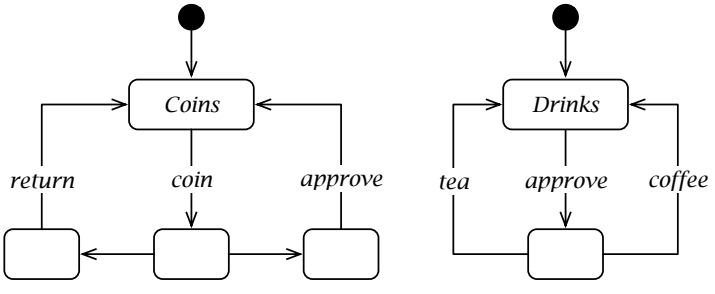


Fig. 4. Vending machine components

Using the two components introduced above, we may produce a description of a vending machine as a parallel combination. The process defined by

$$CVM \;=\; Coins \parallel Drinks$$

is initially ready to engage in *coin*, and will then decide internally whether to accept the coin, and offer a drink, or return it to the user.

We can use the step law for ‖, together with the Law 56, to rewrite this process in sequential form.

$$Coins \parallel Drinks = coin \rightarrow ((Approve \parallel Drinks) \sqcap (Return \parallel Drinks))$$

$$Approve \parallel Drinks = approve \rightarrow (Coins \parallel Ready)$$

$$Return \parallel Drinks = return \rightarrow (Coins \parallel Drinks)$$

$$\begin{aligned} Coins \parallel Ready = {} & coin \rightarrow ((Approve \parallel Ready) \sqcap (Return \parallel Ready)) \,\Box \\ & tea \rightarrow (Coins \parallel Drinks) \,\Box \\ & coffee \rightarrow (Coins \parallel Drinks) \end{aligned}$$

$$\begin{aligned} Approve \parallel Ready = {} & tea \rightarrow (Approve \parallel Drinks) \,\Box \\ & coffee \rightarrow (Approve \parallel Drinks) \end{aligned}$$

$$\begin{aligned} Return \parallel Ready = {} & tea \rightarrow (Return \parallel Drinks) \,\Box \\ & coffee \rightarrow (Return \parallel Drinks) \,\Box \\ & return \rightarrow (Coins \parallel Ready) \end{aligned}$$

It should be obvious that the sequential equivalent of a parallel description may be difficult to construct or understand, whether it is written in process notation as here, or in graphical form: see Figure 5. The use of variable parameters to represent state information (Section 6) will help in this regard.

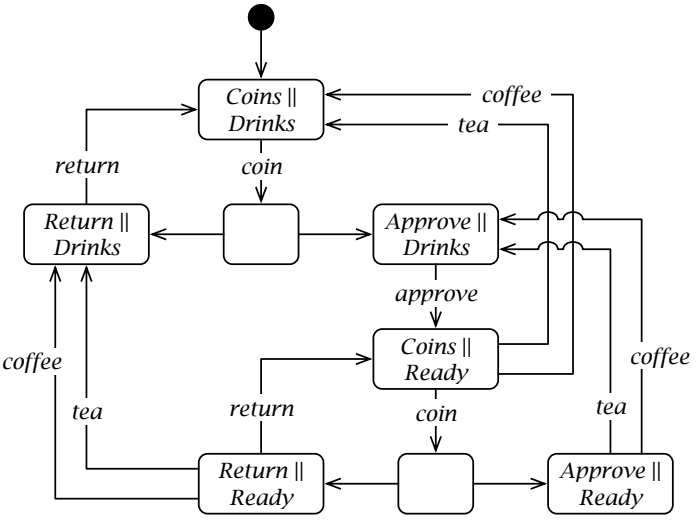


Fig. 5. A sequential view of the parallel combination

If a choice is internal to a particular component, it will be internal to any parallel combination that the component is placed in. In the special case that two components present choices across the *same set of alternatives*: a parallel combination of two external choices presents an external choice to the environment; a parallel combination of external and internal choices behaves as an internal choice; a parallel combination of two internal choices behaves as an internal choice that may deadlock.

Example 33 (Agreement). A group of people are trying to reach agreement upon whether to go out or stay in. At a (very) abstract level, we might model the interaction between them in terms of just two events: *goOut* and *stayIn*, representing agreement to go out, or agreement to stay in, respectively. The behaviour of a helpful, obliging person might be modelled using external choice, whereas the behaviour of someone who will not be influenced by the arguments of others might be modelled using internal choice.

$$\alpha Helpful = \{stayIn, goOut\}$$

$$Helpful = stayIn \rightarrow Skip \,\Box\, goOut \rightarrow Skip$$

$$\alpha Awkward = \{stayIn, goOut\}$$

$$Awkward = stayIn \rightarrow Skip \sqcap goOut \rightarrow Skip$$

The step law for $\|$ confirms that the parallel combination of two helpful people is, again, entirely helpful.

$$Helpful \parallel Helpful = Helpful$$

Applying the distributive law for $\|$, and then the step law to each of the alternatives, confirms that

$$Helpful \parallel Awkward = Awkward$$

Two applications of the distributive law, and four of the step law, together with the fact that $\sqcap$ is idempotent, confirm that

$$Awkward \parallel Awkward = Awkward \sqcap Stop$$

5 Abstraction

When an event appears in a process description, the default assumption is that it represents a single transaction, a point of synchronisation between all of the processes involved—those for which it appears in the corresponding process alphabet—and the environment. In this section, we will introduce three operators that allow us to vary this assumption.

5.1 Interleaving

We use the *name* of an event, in combination with the notion of *alphabets* in parallel composition, to identify the processes whose cooperation is required. In an abstract description, we might wish to use two events with the same name to describe two different transactions; when we do this, it is quite possible that two or more processes could perform the same event independently.

If P and Q are processes, and S is a set of events, then the partially interleaved composition

$$P \,[\![\, S \,]\!]\, Q$$

describes a component that behaves as the parallel composition of P and Q, except that either process may perform events *outside* the shared set S without reference to the other.

Example 34 (Vending machines). We might imagine two different kinds of customer for the vending machine: a tea drinker, and a coffee drinker. Both are able to insert coins, accept drinks, and engage in conversation: an event represented by the event *talk*. Neither will talk properly between inserting a coin and removing a drink: this does not count as engaging in conversation, at least for the purposes of this example.

$$\begin{aligned} TCD &= talk \rightarrow TCD \;\Box \\ &\quad\; coin \rightarrow coffee \rightarrow TCD \end{aligned}$$

$$\begin{aligned} TTD &= talk \rightarrow TTD \;\Box \\ &\quad\; coin \rightarrow tea \rightarrow TTD \end{aligned}$$

The event *coin* appears in both process descriptions, but the two occurrences are intended to represent separate transactions—we have simply chosen not to distinguish between them. Although the two processes must agree upon *talk* events, they can perform *coin* events independently.

$$Customers \;=\; TCD \,[\![\, \{talk\} \,]\!]\, TTD$$

When we use the interleaving operator to construct a parallel composition, the resulting process may be nondeterministic: if an event occurs that is possible for more than one component, but is outside the shared set, then we may be unable to determine which component was involved.

Step Law ($[\![\_]\!]$). If processes P and Q are such that

$$P = \Box\, e : A \bullet e \rightarrow P(e)$$

$$Q = \Box\, e : B \bullet e \rightarrow Q(e)$$

and S is a set of events, then

$$\begin{aligned} P \,[\![\, S \,]\!]\, Q \;=\; & \Box\, e : A \setminus (B \cup S) \bullet e \rightarrow (P(e) \,[\![\, S \,]\!]\, Q) \\ & \Box \\ & \Box\, e : B \setminus (A \cup S) \bullet e \rightarrow (P \,[\![\, S \,]\!]\, Q(e)) \\ & \Box \\ & \Box\, e : A \cap B \cap S \bullet e \rightarrow (P(e) \,[\![\, S \,]\!]\, Q(e)) \\ & \Box \\ & \Box\, e : (A \cap B) \setminus S \bullet e \rightarrow ((P(e) \,[\![\, S \,]\!]\, Q) \sqcap (P \,[\![\, S \,]\!]\, Q(e))) \end{aligned}$$

Example 35 (Vending machines). In combination, the two drinkers may engage in either *talk* or *coin*. If the *coin* event occurs, the outcome is nondeterministic: we do not know whether *tea* or *coffee* is expected.

$$\begin{aligned} TCD \,[\![\, \{talk\} \,]\!]\, TTD = & talk \rightarrow (TCD \,[\![\, \{talk\} \,]\!]\, TTD) \\ & \Box \\ & coin \rightarrow ((coffee \rightarrow TCD \,[\![\, \{talk\} \,]\!]\, TTD) \\ & \qquad \sqcap \\ & \qquad (TCD \,[\![\, \{talk\} \,]\!]\, tea \rightarrow TTD)) \end{aligned}$$

If the shared set is empty—if every common event is taken to denote separate transactions—then we may write $|||$ in place of $[\![\varnothing]\!]$.

Example 36 (Vending machines). The parallel composition of two identical vending machines—operating independently—would be written as

$$PVM \;=\; (VM \,|||\, VM)$$

It is worth observing that this is quite different from the synchronized parallel combination:

$$\begin{aligned} VM \,|||\, VM \;=\; & coin \rightarrow (coin \rightarrow ((coffee \rightarrow VM \,\Box\, tea \rightarrow VM) \\ & \qquad\qquad |||\ (coffee \rightarrow VM \,\Box\, tea \rightarrow VM)) \\ & \Box \\ & coffee \rightarrow PVM \\ & \Box \\ & tea \rightarrow PVM) \end{aligned}$$

whereas

$$\begin{aligned} VM \parallel VM \;=\; & coin \rightarrow (coffee \rightarrow (VM \parallel VM) \\ & \Box \\ & tea \rightarrow (VM \parallel VM) \end{aligned}$$

which is exactly the behaviour described by a single copy of VM.

An interleaved parallel composition can terminate only when all of the component processes have finished.

Law 57 (|||–*Skip*).

$$Skip \mid\mid\mid Skip = Skip$$

Example 37 (Breakfast). The behaviour of the *ParallelBreakfast* process would be unchanged were it to be defined using the interleaving parallel operator:

$$ParallelBreakfast = Juice \mid\mid\mid Coffee \mid\mid\mid Cereal$$

5.2 Renaming

An alternative means of changing the set of events upon which a process is required to synchronize—its alphabet—is afforded by the renaming operator: for any process P,

$$P[a \leftarrow b]$$

describes a component that behaves exactly as P, except that it is ready to perform event b whenever P was ready to perform a: that is, any instance of a within P is shared externally under the name b.

If a renaming operation corresponds to a non-injective function—that is, if two or more source events are renamed to the same target—then the resulting description may be more nondeterministic: we may not be able to tell which of the renamed events has been performed.

Step Law ($\_[\_ \leftarrow \_]$). If process P is such that

$$P = \Box\, e : A \bullet e \rightarrow P(e)$$

and a and b are events, then

$$\begin{aligned} P[a \leftarrow b] = {} & \Box\, e : A \setminus \{a, b\} \bullet e \rightarrow P(e)[a \leftarrow b] \\ & \Box \\ & \Box\, e : A \cap \{a\} \bullet b \rightarrow P(a)[a \leftarrow b] \\ & \Box \\ & \Box\, e : A \cap \{b\} \bullet b \rightarrow P(b)[a \leftarrow b] \end{aligned}$$

Example 38 (Breakfast). If we use the same name for the action of obtaining a *cup* and the action of obtaining a *glass*, then we may not know whether the parallel breakfast process is ready to accept *juice* or *coffee*.

$$
\begin{aligned}
ParallelBreakfast[glass \leftarrow cup] \;=\; & cup \rightarrow (juice \rightarrow \ldots \;\Box \\
& \qquad\qquad bowl \rightarrow \ldots \;\Box \\
& \qquad\qquad cup \rightarrow \ldots\,) \\
& \Box \\
& cup \rightarrow (cup \rightarrow \ldots \;\Box \\
& \qquad\qquad bowl \rightarrow \ldots \;\Box \\
& \qquad\qquad coffee \rightarrow \ldots\,) \\
& \Box \\
& bowl \rightarrow (cup \rightarrow \ldots \;\Box \\
& \qquad\qquad cereal \rightarrow \ldots \;\Box \\
& \qquad\qquad cup \rightarrow \ldots\,)
\end{aligned}
$$

Renaming distributes through the sequential operators ($\rightarrow$, $\Box$, $\sqcap$, and ;) but not, in general, through parallel composition. A non-injective renaming may give two separate, unshared events the same name. A parallel combination of renamed components would share the renamed event internally; a renamed parallel combination would not.

Example 39 (Boxes). If we were to represent both the red and blue boxes simply as *small* boxes (and the *green* box as *large*), then we should still expect to see three boxes loaded, with the large box being either the first or the second box to be loaded.

$$
\begin{aligned}
& (Matthew \parallel Marina)[red \leftarrow small, blue \leftarrow small, green \leftarrow large] \\
& \\
& \quad = \; small \rightarrow large \rightarrow small \rightarrow Stop \\
& \qquad \Box \\
& \qquad large \rightarrow small \rightarrow small \rightarrow Stop
\end{aligned}
$$

If we were to rename the component processes, and then combine them in the same way as before, we would obtain a different, incorrect result:

$$
\begin{aligned}
& Matthew[red \leftarrow small, blue \leftarrow small, green \leftarrow large] \\
& \parallel \\
& Marina[red \leftarrow small, blue \leftarrow small, green \leftarrow large] \\
& \\
& \quad = \; large \rightarrow small \rightarrow Stop
\end{aligned}
$$

This use of parallel combination is inconsistent with the intended interpretation of the event names. If we wished to combine the renamed components, we would need to use the partial interleaving operator:

$$
\begin{aligned}
& Matthew[red \leftarrow small, blue \leftarrow small, green \leftarrow large] \\
& |[\{large\}]| \\
& Marina[red \leftarrow small, blue \leftarrow small, green \leftarrow large] \\
& \\
& \quad = \; small \rightarrow large \rightarrow small \rightarrow Stop \\
& \qquad \Box \\
& \qquad large \rightarrow small \rightarrow small \rightarrow Stop
\end{aligned}
$$

5.3 Hiding

If P is a process, and A is a set of events, then the process $P \setminus H$ describes a component that behaves exactly as P, except that events from H are *hidden*: they are no longer observed at the interface to the component.

Example 40 (Vending machines). The *approve* event was intended as an internal communication between the two components of the machine CVM: we might hide it to obtain a more complete description of its behaviour. The resulting process $CVM \setminus \{approve\}$ could be rewritten as follows.

$$
\begin{aligned}
&(Coins \parallel Drinks)' \;= \\
&\qquad coin \rightarrow ((Coins \parallel Ready)' \sqcap (Return \parallel Drinks)')
\end{aligned}
$$

$$
\begin{aligned}
&(Return \parallel Drinks)' \;= \\
&\qquad return \rightarrow (Coins \parallel Drinks)'
\end{aligned}
$$

$$
\begin{aligned}
&(Coins \parallel Ready)' \;= \\
&\qquad coin \rightarrow ((Approve \parallel Ready)' \sqcap (Return \parallel Ready)') \;\Box \\
&\qquad tea \rightarrow (Coins \parallel Drinks)' \;\Box \\
&\qquad coffee \rightarrow (Coins \parallel Drinks)'
\end{aligned}
$$

$$
\begin{aligned}
&(Approve \parallel Ready)' \;= \\
&\qquad tea \rightarrow (Coins \parallel Ready)' \;\Box \\
&\qquad coffee \rightarrow (Coins \parallel Ready)'
\end{aligned}
$$

$$
\begin{aligned}
&(Return \parallel Ready)' \;= \\
&\qquad tea \rightarrow (Return \parallel Drinks)' \;\Box \\
&\qquad coffee \rightarrow (Return \parallel Drinks)' \;\Box \\
&\qquad return \rightarrow (Coins \parallel Ready)'
\end{aligned}
$$

The stage known as $Approve \parallel Drinks$ is now identified with $Coins \parallel Ready$: no external events were possible in this state, and now-internal event *approve* leads directly to $Coins \parallel Ready$ (see Figure 6).

Hiding events will often increase the degree of nondeterminism in a process description. At any point where a hidden event is possible, we will not know whether or not it has occurred. In this case, the behaviour of a component will be described by an internal choice: one alternative will describe the behaviour of the component if the hidden event has not been performed; the other, the behaviour if it has.

Step Law ($\setminus$). If process P is such that

$$P = \Box\, e : A \bullet e \rightarrow P(e)$$

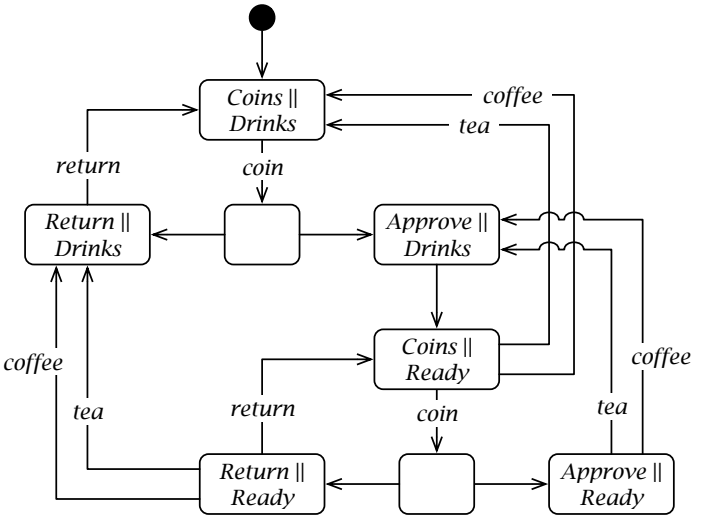


Fig. 6. Hiding the *approve* event

and H is any set of events, then

$$\begin{aligned} P \setminus H \;=\; & \sqcap h : (A \cap H) \bullet P(h) \setminus H \\ & \sqcap \\ & (\,\Box\, e : (A \setminus H) \bullet e \rightarrow P(e) \setminus H \\ & \;\Box \\ & \;\sqcap h : (A \cap H) \bullet P(h) \setminus H\,) \end{aligned}$$

Any hidden event (any event in $A \cap H$) may have occurred, and so we may be presented with an indexed internal choice of consequent processes.

$$\sqcap h : (A \cap H) \bullet P(h) \setminus H$$

Alternatively, if no event has occurred, then the process presents an external choice over the remaining, visible part of the menu ($A \setminus H$), together with the option of simply allowing an internal event to occur, and thus being presented with the resulting indexed internal choice.

As we might expect, the hiding operator is distributive.

Law 58. *For any processes P and Q, and set of events H,*

$$(P \sqcap Q) \setminus H \;=\; (P \setminus H) \sqcap (Q \setminus H)$$

Example 41 (Vending machines). We may simplify our description of the vending machine by deciding to ignore the occurrence or availability of the *return* event; the behaviour of the process $CVM \setminus \{return\}$ is described by

$$
\begin{aligned}
(Coins \parallel Drinks)'' &= coin \rightarrow ((Approve \parallel Drinks)'' \sqcap (Coins \parallel Drinks)'') \\
(Approve \parallel Drinks)'' &= approve \rightarrow (Coins \parallel Ready)'' \\
(Coins \parallel Ready)'' &= coin \rightarrow ((Approve \parallel Ready)'' \sqcap (Return \parallel Ready)'' \sqcap \\
&\qquad (Coins \parallel Ready)'') \ \Box \\
&\quad tea \rightarrow (Coins \parallel Drinks)'' \ \Box \\
&\quad coffee \rightarrow (Coins \parallel Drinks)''
\end{aligned}
$$

$$
\begin{aligned}
(Approve \parallel Ready)'' &= tea \rightarrow (Approve \parallel Drinks)'' \ \Box \\
&\quad coffee \rightarrow (Approve \parallel Drinks)'' \\
(Return \parallel Ready)'' &= tea \rightarrow (Return \parallel Drinks)'' \ \Box \\
&\quad coffee \rightarrow (Return \parallel Drinks)''
\end{aligned}
$$

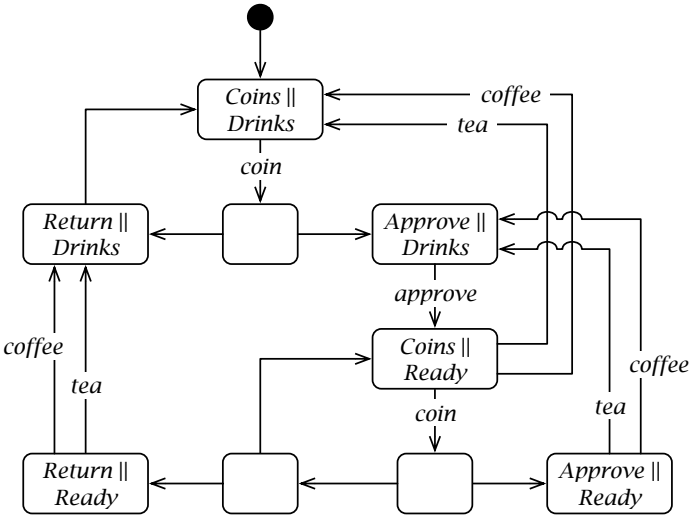


Fig. 7. Hiding the *return* event

Figure 7 shows how the insertion of a coin at the initial stage $Coins \parallel Drinks$ may make no difference at all to the future behaviour: the internal choice could be resolved in favour of $Return \parallel Drinks$, which is now indistinguishable from $Coins \parallel Drinks$.

The insertion of a coin at the stage $Coins \parallel Ready$ might also make no difference: as before, the future behaviour could be that of $Approve \parallel Ready$ or (what we have still decided to call) $Return \parallel Ready$, but it could also be that of $Coins \parallel Ready$.

Hiding too many events in a process description can render it useless. Our processes are defined in terms of the occurrence and availability of external, observable events; a recursive definition is valid only if each recursive instance of the process name being defined is guarded by at least one event. When we hide

events that are serving as guards, we may find that the resulting behaviour is chaotic; in hiding events, we may be introducing *divergences*.

Example 42 (Vending machines). If we hide both *coin* and *return*, then we might hope to obtain a simpler description of the vending machine sufficient for the analysis of properties involving *tea*, *coffee*, and *approve*.

For example, we might hope to establish that a choice between tea and coffee is offered every time that an approve event occurs, by defining a process

$$\begin{aligned} \mathit{Offer} &= \mathit{approve} \rightarrow \mathit{Choice} \\ \mathit{Choice} &= (\mathit{tea} \rightarrow \mathit{Offer} \,\Box\, \mathit{coffee} \rightarrow \mathit{Offer}) \end{aligned}$$

and checking that

$$\mathit{Offer} \sqsubseteq_{FD} (\mathit{CVM} \setminus \{\mathit{coin}, \mathit{return}\})$$

However, this check will fail: the empty trace $\langle\rangle$ appears in the divergences set $\mathit{divergences}\,[\![\mathit{CVM} \setminus \{\mathit{coin}, \mathit{return}\}]\!]$ but not in $\mathit{divergences}\,[\![\mathit{Offer}]\!]$. The behaviour at the initial stage is defined by an unguarded recursion:

$$\begin{aligned} \mathit{Coins} \parallel \mathit{Drinks} &= \mathit{Return} \parallel \mathit{Drinks} \\ \mathit{Return} \parallel \mathit{Drinks} &= \mathit{Coins} \parallel \mathit{Drinks} \end{aligned}$$

There is not enough information here to determine a useful failure semantics.

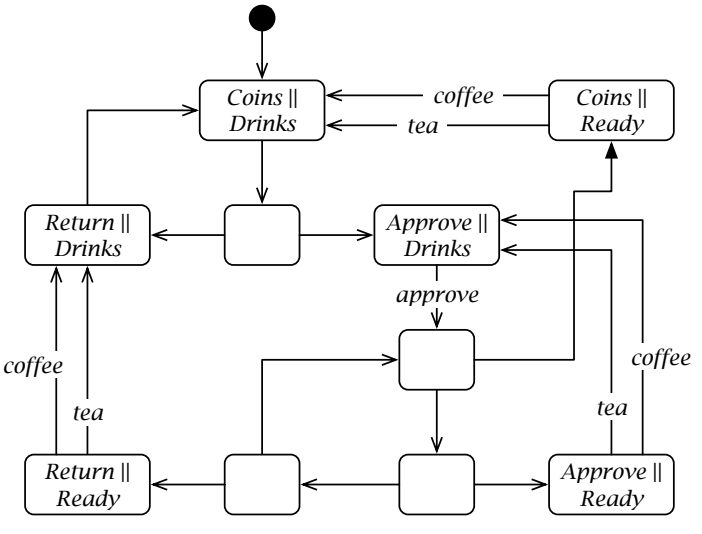


Fig. 8. Hiding both *coin* and *return*

The same is true following the occurrence of an *approve* event: there are three unlabelled stages at this point—see Figure 8—at which the behaviour is defined by an unguarded recursion; the loop of unlabelled transitions marks another region of chaos in the process description.

If hiding a set of events would introduce divergences, and render our intended analysis impossible, then we have three choices as to how to proceed: hide the events, but restrict our analysis to properties that can be checked using the trace semantics; do not hide the events, and rewrite any processes used to characterize properties so that these events are ignored; and hide the events, but only after introducing a constraining *assumption* process.

Example 43 (Vending machines). If we are content to check the weaker property that the *approve* and drinks (*tea* or *coffee*) events occur alternately, if they occur at all—then we could do this using the traces model, by defining

$$Alternate \;=\; approve \rightarrow (coffee \rightarrow Alternate \sqcap tea \rightarrow Alternate)$$

and checking the refinement

$$Alternate \sqsubseteq_T CVM \setminus \{coin, return\}$$

Example 44 (Vending machines). If we wish to check the property that a choice between *tea* and *coffee* is offered every time that an *approve* event occurs, then we cannot hide the events *coin* and *return*—for we would lose all availability information. Instead, we define a process that expresses this property in the presence of those events.

$$\begin{aligned} Offer' \;&=\; (approve \rightarrow Choice) \\ &\quad\;\Box \\ &\quad\; (coin \rightarrow Offer' \sqcap return \rightarrow Offer' \sqcap Stop) \end{aligned}$$

$$\begin{aligned} Choice' \;&=\; (tea \rightarrow Offer' \;\Box\; coffee \rightarrow Offer') \\ &\quad\;\Box \\ &\quad\; (coin \rightarrow Choice' \sqcap return \rightarrow Choice' \sqcap Stop) \end{aligned}$$

Considered as a specification, the process *Offer′* does not require that *coin* or *return* is offered at any stage; and if either event occurs, it has no effect upon the offer of *approve*, or the choice between *tea* and *coffee*.

This is a more complex description than *Offer*, as we were forced to consider other events, but it allows us to confirm that the required property holds—by checking the following refinement:

$$Offer' \;\sqsubseteq\; CVM$$

Example 45 (Vending machines). We may write a process to exclude the problematic behaviours—in which a coin is inserted, and the return button pressed, and this cycle is repeated indefinitely—from consideration. The process defined by

$$\begin{aligned} Assumption &= return \rightarrow Choose \ \Box \\ &\quad\ tea \rightarrow Assumption \ \Box \\ &\quad\ coffee \rightarrow Assumption \end{aligned}$$

$$\begin{aligned} Choose &= tea \rightarrow Assumption \ \Box \\ &\quad\ coffee \rightarrow Assumption \end{aligned}$$

allows *return* to occur, but insists that one of *tea* or *coffee* must be performed before it can occur again.

Placed in parallel with the description of the vending machine, using the alphabet $\alpha Assumption = \{return, tea, coffee\}$ this process will have the required effect of excluding the problematic behaviours: the process

$$(CVM \parallel Assumption) \setminus \{coin, return\}$$

has no divergences, and we can confirm that the property holds, using the refinement check:

$$Offer \quad \sqsubseteq_{FD} \quad (CVM \parallel Assumption) \setminus \{coin, return\}$$

If this assumption seems too restrictive, then we may use a value parameter—see the next section—to count occurrences of *return*, and choose a higher limit upon the number of times that it can occur between two consecutive occurrences of *tea* or *coffee*.

6 Data

The process language of CSP is augmented by a language of datatypes: sets, functions, and sequences. The names of processes and events may be parameterized using values from sets of any type, including sets of events. Indeed, we have already seen an example of parameterisation: in our use of indexed external choice, the consequence of the choice was described by a process parameterized by a variable denoting the chosen event.

6.1 Process Parameters

We may use parameters in process names to record state information: representing the stages of a process in terms of the value of a collection of variables. These values may change with the performance of events: expressions determining the new values may be used to parameterize instances of the process name appearing on the right-hand side of the defining equations.

If a process name includes a state parameter, it is better to enclose it within a local definition environment: if D is a collection of declarations—of sets, functions, or processes—then the expression

$$\mathsf{let}\ D\ \mathsf{within}\ E$$

has the value of E within the additional context provided by D.

Example 46 (Lift). We may use the following process to describe the behaviour of a simple lift, moving up and down between three floors:

$$
\begin{array}{l}
\mathit{Lift} \;=\; \mathsf{let} \\
\qquad \mathit{LiftAtFloor}(0) = \mathit{up} \rightarrow \mathit{LiftAtFloor}(1) \\
\qquad \mathit{LiftAtFloor}(1) = \mathit{down} \rightarrow \mathit{LiftAtFloor}(0) \\
\qquad\qquad\qquad\qquad \Box \\
\qquad\qquad\qquad\qquad \mathit{up} \rightarrow \mathit{LiftAtFloor}(2) \\
\qquad \mathit{LiftAtFloor}(2) = \mathit{down} \rightarrow \mathit{LiftAtFloor}(1) \\
\quad \mathsf{within} \\
\qquad \mathit{LiftAtFloor}(0)
\end{array}
$$

This process can perform the event *up* at stages *0* and *1*, corresponding to the lift being at Floors *0* or *1*, and *down* at stages *1* and *2*.

We may use a conditional syntax to represent several definitions, or stages, within a single defining equation. If B is a Boolean-valued expression, then the value of

$$\text{if } B \text{ then } X \text{ else } Y$$

is that of X if B is true, and Y otherwise.

Example 47 (Lift). A lift that can move between Floors 0 and 9 could be described by the following process:

$$
\begin{array}{l}
\mathit{Lift} \;=\; \mathsf{let} \\
\qquad \mathit{LiftAtFloor}(n) \;= \\
\qquad\quad \text{if } n = 0 \text{ then} \\
\qquad\qquad \mathit{up} \rightarrow \mathit{LiftAtFloor}(1) \\
\qquad\quad \text{else if } n \in 1 \mathinner{..} 8 \text{ then} \\
\qquad\qquad \mathit{down} \rightarrow \mathit{LiftAtFloor}(n-1) \\
\qquad\qquad \Box \\
\qquad\qquad \mathit{up} \rightarrow \mathit{LiftAtFloor}(n+1) \\
\qquad\quad \text{else} \\
\qquad\qquad \mathit{down} \rightarrow \mathit{LiftAtFloor}(8) \\
\quad \mathsf{within} \\
\qquad \mathit{LiftAtFloor}(0)
\end{array}
$$

Here, variable n is declared on the left of the equation, and corresponds to the number of the current floor.

If the conditional expression denotes a process, and that process is to appear in an external choice, then we may find the following abbreviation useful: if B is a Boolean-valued expression, and P is a process, then we write

$$B \mathbin{\&} P$$

to denote the conditional expression

$$\text{if } B \text{ then } P \text{ else } \mathit{Stop}$$

As the process *Stop* has no effect upon the other components of an external choice (it is akin to an option on a menu being "greyed-out") the expression B serves as a *guard*, or enabling condition, for alternative P. For example, if B and C are Boolean-valued expressions, and P and Q are processes, then

$$\begin{array}{l} B \mathbin{\&} P \\ \Box \\ C \mathbin{\&} Q \end{array}$$

is a more convenient way of presenting the conditional expression

$$\begin{array}{l} \text{if } B \wedge C \text{ then} \\ \quad P \Box Q \\ \text{else if } B \wedge \neg\, C \text{ then} \\ \quad P \\ \text{else if } C \wedge \neg\, B \text{ then} \\ \quad Q \\ \text{else} \\ \quad \mathit{Stop} \end{array}$$

Example 48 (Lift). We may use guarded process expressions to describe the behaviour of the lift moving between Floors 0 and 9:

$$\begin{array}{l} \mathit{Lift} \;=\; \mathsf{let} \\ \qquad\qquad \mathit{LiftAtFloor}(n) \;= \\ \qquad\qquad\quad n \leq 8 \mathbin{\&} \mathit{up} \rightarrow \mathit{LiftAtFloor}(n+1) \\ \qquad\qquad\quad \Box \\ \qquad\qquad\quad n \geq 1 \mathbin{\&} \mathit{down} \rightarrow \mathit{LiftAtFloor}(n-1) \\ \qquad\; \mathsf{within} \\ \qquad\qquad \mathit{LiftAtFloor}(0) \end{array}$$

We may also wish to parameterize process names in order to identify particular instances of a generic process. If we employ the local declaration syntax, then the two kinds of process parameter need never meet: instance (or static) parameters keep the same value for the lifetime of the process, and appear as part of the main, external process name.

Example 49 (Lift). A generic lift process, able to move between Floor 0 and Floor max, could be described by the following process:

$$
\begin{array}{l}
GenericLift(max) \;=\; \mathsf{let} \\
\qquad\qquad LiftAtFloor(n) \;= \\
\qquad\qquad\qquad n \leq max \;\&\; up \rightarrow LiftAtFloor(n+1) \\
\qquad\qquad\qquad \Box \\
\qquad\qquad\qquad n \geq 1 \;\&\; down \rightarrow LiftAtFloor(n-1) \\
\qquad\quad \mathsf{within} \\
\qquad\qquad LiftAtFloor(0)
\end{array}
$$

The value of parameter max does not change during the lifetime of an instance of this process. For example,

$$GenericLift(9) \;=\; Lift$$

where $Lift$ is the process describing a lift moving between 10 floors, defined in the previous examples.

6.2 Compound Events

We may use parameters in event names to indicate an event that involves a particular instance of a generic process, or to represent data being passed between processes in a parallel combination. In either case, each parameter will appear after a single dot in the process name.

Example 50 (Car doors). A generic process representing the behaviour of a car door could be described as

$$\alpha SimpleDoor(i) \;=\; \{open.i, close.i, lock.i, unlock.i\}$$

$$
\begin{array}{l}
SimpleDoor(i) \;=\; \mathsf{let} \\
\qquad\qquad Open \;=\; close.i \rightarrow Closed \\
\qquad\qquad Closed \;=\; open.i \rightarrow Open \\
\qquad\qquad\qquad\qquad \Box \\
\qquad\qquad\qquad\qquad lock.i \rightarrow unlock.i \rightarrow Closed \\
\qquad\quad \mathsf{within} \\
\qquad\qquad Open
\end{array}
$$

The process $SimpleDoor(1)$ is capable of engaging in events from the set

$$\{open.1, close.1, lock.1, unlock.1\}$$

It can perform these events independently of the other three processes in the following parallel combination

$$CarDoors \;=\; \|\, k : 1 \,..\, 4 \bullet SimpleDoor(k)$$

A set of parameterized events with the same basic name, but with different values for one or more of the parameters, is called a *channel*. A single channel may be shared by several processes—all parties to the same transaction—with each process able to constrain the values of the various parameters of any event that occurs.

Example 51 (Meeting). A meeting event has three parameters: location, date, and time of day. A typical occurrence of such an event might be written as

$$meeting.london.15/03/2006.am$$

The behaviour of three people who are attempting to arrange a meeting may be described in part by the following processes:

$$Steve = \Box\, l : \{london\};\ d : Date;\ t : TimeOfDay \bullet meeting.l.d.t \rightarrow \ldots$$

$$Fiona = \Box\, l : Location;\ d : \{14/03/2006\};\ t : TimeOfDay \bullet meeting.l.d.t \rightarrow \ldots$$

$$James = \Box\, l : Location;\ d : \{14/03/2006, 16/03/2006\};\ t : \{am, lunch\} \bullet meeting.l.d.t \rightarrow \ldots$$

Steve can attend a meeting on any of the possible dates, at any time of day, provided that it is in London. Fiona can attend only on the 14th of March, but is available at any time on that day. Like Fiona, James is happy with any location, but is available only for morning or lunchtime meetings on the 14th or the 16th.

The parallel combination of these three processes, all sharing the channel *meeting*, has the same initial behaviour as the following external choice:

$$\begin{aligned} &Steve \parallel Fiona \parallel James \\ &\quad = \Box\, l : \{london\};\ d : \{14/03/2006\};\ t : \{am, lunch\} \bullet \\ &\qquad meeting.l.d.t \ldots \\ &\quad = meeting.london.14/03/2006.am \rightarrow \ldots \\ &\qquad \Box \\ &\qquad meeting.london.14/03/2006.lunch \rightarrow \ldots \end{aligned}$$

If a process is ready to perform a parameterized event for several values of a particular parameter, then we may represent that part of the external choice by including a question mark at the appropriate point in the event name. For any process P and parameterized event c, we define

$$c\,?(x : X) \rightarrow P \;=\; \Box\, x : X \bullet c.x \rightarrow P$$

If the range T describes all possible values of the parameter x, then we write $c\,?(x : T)$ simply as $c\,?x$. If the range is a singleton set, containing expression e, then we write $c\,?(x : \{e\})$ simply as $c\,!e$.

Example 52 (Meeting). The behaviour of the three people attempting to arrange a meeting could be described in abbreviated form as:

$$Steve \;=\; meeting\,!london\,?d\,?t \rightarrow \ldots$$

$$Fiona \;=\; meeting\,?l\,!14/03/2006\,?t \rightarrow \ldots$$

$$James \;=\; meeting\,?l\,?(d : \{14/03/2006, 16/03/2006\})\,?(t : \{am, lunch\}) \rightarrow \ldots$$

6.3 Message Passing

If a channel of parameterized events is shared between exactly two processes, then it can be used to describe message-passing communication between them. In most cases, one process will determine the value of any parameters, and the other will be ready to accept any combination. A special case of the step law for parallel composition shows how this corresponds to values being passed.

Law 59. *If processes P and Q are such that*

$$P{=}c!e \rightarrow P'$$
$$Q{=}c?x \rightarrow Q(x)$$

where e is an appropriately-valued expression, then

$$P \parallel Q \;=\; c!e \rightarrow (\,P' \parallel Q(e)\,)$$

Following the shared event, the future behaviour of Q may be determined by the value of e, originally 'known only' to P.

Example 53 (Vending machines). The following process describes the behaviour of a machine that accumulates coins, and will dispense selected drinks if the current holding exceeds a fixed price.

$$\begin{array}{l}
SVM(price) \;= \\
\quad \mathsf{let} \\
\qquad Ready(holding) \;=\; insertCoin?value \rightarrow \text{if } holding + value < price \text{ then} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad Ready(holding + value) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{else} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad Select(holding + value) \\
\\
\qquad Select(holding) \;=\; insertCoin?value \rightarrow Select(holding + value) \\
\qquad\qquad\qquad\qquad\quad \Box \\
\qquad\qquad\qquad\qquad\quad select?x \rightarrow \text{if } holding < 2 * price \text{ then} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad dispense!x \rightarrow Ready(holding - price) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{else} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad dispense!x \rightarrow Select(holding - price) \\
\quad \mathsf{within} \\
\qquad Ready(0)
\end{array}$$

This is a process representation of the state machine shown in Figure 2.

The behaviour of a particular user, who will insert a 20 pence or a 50 pence coin, select an orange drink, and then collect whatever drink is dispensed, might be described by

$$User \;=\; \sqcap\, c : \{20, 50\} \bullet \\ insertCoin!c \rightarrow select!orange \rightarrow dispense?x \rightarrow Stop$$

The possible interactions between this process and the vending machine (with *price* set to 50) are described by the parallel combination

$$User \parallel SVM(50) \;= \\ insertCoin.20 \rightarrow Stop \\ \sqcap \\ insertCoin.50 \rightarrow select.orange \rightarrow dispense.orange \rightarrow Stop$$

This process is consistent with—but does not guarantee—the use case shown in the sequence diagram of Figure 1.

A single channel is an adequate representation of message-passing communication only if the sending of a message is synonymous with its reception. If it matters that a component may send a message without knowing that it has been received, or that some event may be performed by another component while the message is being passed, then we may need to use two channels, and an intervening process, to model the communication.

7 Communication

One class of problems in distributed computing concerns the provision of reliable communication across possibly unreliable media. In addressing these problems, we may use processes to describe the service that we aim to provide, and check that these are refined by a combination of processes representing the protocol components and the media that they are intended to operate over.

7.1 Buffers

In most cases, our notion of a reliable service can be described in terms of *buffers*: processes that store and forward messages in such a way that no messages are lost, and the order of messages is preserved. If we use a channel *send* to represent the sending of messages, and *receive* to represent their reception, then this corresponds to the constraint that, in any behaviour of a buffer process:

- the sequence of values observed as parameters to *receive* events is an initial subsequence of those observed as parameters to *send*;
- if these two sequences are not equal, then an appropriate *receive* event is available;
- if these two sequences *are* equal—so that there are no messages 'in flight', then no *send* event can be refused.

This constraint upon behaviours is completely characterized by the requirement that the process should be a refinement of the following process.

$$
\begin{array}{l}
\mathit{Buffer} \;=\; \mathsf{let} \\
\qquad \mathit{State}(s) \;=\; \text{if } s = \langle\rangle \text{ then} \\
\qquad\qquad\qquad\qquad \mathit{send}\,?x \rightarrow \mathit{State}(\langle x\rangle) \\
\qquad\qquad\qquad \text{else} \\
\qquad\qquad\qquad\qquad \mathit{send}\,?x \rightarrow \mathit{State}(s \frown \langle x\rangle) \\
\qquad\qquad\qquad\qquad \triangleright \\
\qquad\qquad\qquad\qquad \mathit{receive}\,!\mathit{head}(s) \rightarrow \mathit{State}(\mathit{tail}(s)) \\
\quad \mathsf{within} \\
\qquad \mathit{State}(\langle\rangle)
\end{array}
$$

This process describes a buffer process of undetermined (and possibly changing) capacity. If the value of state parameter s is the empty sequence, then *send* is possible; if not, then *send* may or may not be possible, but the next *receive* event is definitely available.

If we wish to use existing refinement-checking technology to establish that a process is a buffer, then we will need to compare that process with a finite approximation to *Buffer*.

Example 54 (Buffers). The following process describes a buffer with undetermined capacity no greater than N.

$$
\begin{array}{l}
\mathit{Buffer}(N) \;=\; \mathsf{let} \\
\qquad \mathit{State}(s) \;=\; \text{if } s = \langle\rangle \text{ then} \\
\qquad\qquad\qquad\qquad \mathit{send}\,?x \rightarrow \mathit{State}(\langle x\rangle) \\
\qquad\qquad\qquad \text{else} \\
\qquad\qquad\qquad\qquad (\#s < N)\;\&\; \mathit{send}\,?x \rightarrow \mathit{State}(s \frown \langle x\rangle) \\
\qquad\qquad\qquad\qquad \triangleright \\
\qquad\qquad\qquad\qquad \mathit{receive}\,!\mathit{head}(s) \rightarrow \mathit{State}(\mathit{tail}(s)) \\
\quad \mathsf{within} \\
\qquad \mathit{State}(\langle\rangle)
\end{array}
$$

Here, the *send* event is blocked if there are already N messages in flight. Using the semantic functions for the failures-fivergences model, we can show that

$$\mathit{Buffer} \;\sqsubseteq_{FD}\; \mathit{Buffer}(N)$$

for any value of N greater than 0. (It would not be possible to demonstrate this automatically, using the existing refinement-checking tools, as the state space of *Buffer* would be infinite.)

In many cases, the degree of buffering will be precisely determined, and will remain constant throughout the lifetime of the process. We will then be able to show that the process is a refinement of a deterministic buffer process of some fixed capacity.

Example 55 (Buffers). A buffer of capacity exactly N is described by the following process:

$$\begin{array}{l} Copy(N) \;=\; \mathsf{let} \\ \qquad State(s) \;=\; \#s < N \;\&\; send\,?x \rightarrow State(s \frown \langle x \rangle) \\ \qquad\qquad\qquad \Box \\ \qquad\qquad\qquad \#s > 0 \;\&\; receive\,!head(s) \rightarrow State(tail(s)) \\ \quad \mathsf{within} \\ \qquad State(\langle\rangle) \end{array}$$

The limit upon the capacity of $Buffer(N)$ means that we can establish that

$$Buffer(N) \;\sqsubseteq_{FD}\; Copy(N)$$

through an exhaustive exploration of the state space.

7.2 Protocols

A protocol is a set of rules for collaboration: if each party follows the rules that apply to them, then the collaboration will be successful, in some respect. In the case of a communication protocol, the outcome of a successful collaboration is the provision of a reliable service for transferring data.

We may represent the design of a protocol as a parallel composition of processes representing the intended behaviour of various protocol components like nodes, service access points, clients, servers, senders, or receivers. For example, a data transfer protocol that consists of rules for a *Sender* and *Receiver* process would be described as

$$Protocol \;=\; Sender \parallel Receiver$$

In most cases, the alphabets of the processes representing protocol components will be mutually disjoint. If these processes were to share events, then the design of a reliable communication mechanism would most likely be a trivial matter.

We will define two further processes: one to describe the service provided to the protocol components by the media connecting them, and another to describe the communication service that the protocol itself is intended to provide. If we call these processes *Media* and *Service*, respectively, then our intention will be to show that

$$Service \;\sqsubseteq_{FD}\; (Protocol \parallel Media) \setminus Internal$$

where *Internal* is the set of all events used to describe the protocol implementation (and the supporting media) that do not form part of the description of the intended service.

Example 56 (Protocols). Even if the supporting media provide a reliable service, we may still wish to define a protocol to control the degree of buffering in a

system. We may consider a simple flow control protocol for data communication, in which the behaviour of the sending component is described by

$$
\begin{array}{l}
Sender \;=\; \mathsf{let} \\
\qquad Ready \;=\; in?m \rightarrow Holding(m) \\
\qquad Holding(m) \;=\; sender\_send!m \rightarrow sender\_receive?a \rightarrow Ready \\
\mathsf{within} \\
\qquad Ready
\end{array}
$$

and that of the receiver is described by

$$
\begin{array}{l}
Receiver \;=\; \mathsf{let} \\
\qquad Ready \;=\; receiver\_receive?m \rightarrow Holding(m) \\
\qquad Holding(m) \;=\; out!m \rightarrow receiver\_send!ack \rightarrow Ready \\
\mathsf{within} \\
\qquad Ready
\end{array}
$$

We will suppose that the communication media linking them provide a reliable, buffered service, with the degree of buffering no greater than N:

$$
\begin{array}{l}
Media \;=\; Buffer(N)[send \leftarrow sender\_send, receive \leftarrow receiver\_receive] \\
\qquad \parallel \\
\qquad Buffer(N)[send \leftarrow receiver\_send, receive \leftarrow sender\_receive]
\end{array}
$$

The intended service is that of a one-place buffer, described by

$$Service \;=\; Copy(1)$$

and the set of events 'internal' to the mechanism is given by

$$
\begin{array}{l}
Internal \;=\; \bigcup\{\; m : Message;\; a : Ack \bullet \{\, sender.send.m, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad receiver.receive.m, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad receiver.send.a, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad sender.receive.a \,\} \,\}
\end{array}
$$

For any value of N, we will be able to establish that

$$Service \;\sqsubseteq_{FD}\; (Protocol \parallel Media) \setminus Internal$$

and hence that this design of protocol reduces the degree of buffering to the point where they can be at most one message 'in flight'.

If the service provided by the supporting media is not reliable, then some strategy for message numbering may be required. If the media may lose messages, then the sender should be prepared to retransmit, but transmitting the same message twice introduces the risk of duplicates arriving at the receiver: we require a means of recognising that a particular message has already arrived. If the media may reorder messages, then numbering is essential.

Example 57 (Protocols). A sending process may tag each message with a message sequence number, and increment this number with each new message accepted for transmission.

$$
\begin{array}{l}
Sender' \;=\; \mathsf{let} \\
\quad Ready(num) \;=\; in?m \rightarrow Holding(m, num) \\
\quad Holding(m, num) \;=\; sender\_send!m!num \rightarrow Waiting(m, num) \\
\quad Waiting(m, num) \;=\; sender\_receive?a \rightarrow \\
\qquad\qquad\qquad\qquad Ready((num + 1) \bmod max) \\
\qquad\qquad\qquad\qquad \Box \\
\qquad\qquad\qquad\qquad timeout \rightarrow Holding(m, num) \\
\quad \mathsf{within} \\
\quad\quad Ready(0)
\end{array}
$$

If the sender times out while waiting for an acknowledgement, an action represented by the event *timeout*, then it will retransmit the current message with the same message sequence number. Message sequence numbers are re-used once the modulus value *max* is reached.

A complementary receiver process will discard any message that does not have the expected sequence number—whether it is a duplicate of one already received, or a message with a later number (indicating that an intervening message has been lost).

$$
\begin{array}{l}
Receiver' \;=\; \mathsf{let} \\
\quad Ready(num) \;=\; receiver\_receive?m?n \rightarrow \\
\qquad\qquad\qquad\qquad \mathsf{if}\ n = num\ \mathsf{then} \\
\qquad\qquad\qquad\qquad\quad Holding(m, num) \\
\qquad\qquad\qquad\qquad \mathsf{else} \\
\qquad\qquad\qquad\qquad\quad Ready(num) \\
\quad Holding(m, num) \;=\; out!m \rightarrow receiver\_send!ack \rightarrow \\
\qquad\qquad\qquad\qquad\quad Ready((num + 1) \bmod max) \\
\quad \mathsf{within} \\
\quad\quad Ready(0)
\end{array}
$$

The degree of flow control provided by the design of the sender process means that no reordering will be possible, whatever the service provided by the media, as there can be at most one message in flight at any one time.

This design can be generalized to one in which the sender can store several messages pending acknowledgement, the receiver can store several messages pending the arrival of a missing, earlier message, and each acknowledgement contains a message sequence number. Such a design will cope with unreliable media provided that there is some limit upon the degree of message loss; the sender does not time out too quickly; and there may be no more than *max* messages in flight.

7.3 Assumptions

If the supporting media may lose messages, or if there are timing properties involved, then we may need to make make additional, explicit assumptions about the behaviour of the protocol components. As in Section 5.3, these assumptions may be incorporated by modifying the *Service* property, or by adding additional processes to constrain the behaviour of the design.

Example 58 (Protocols). If the media connecting the sender and receiver were described by the following process

$$
\begin{array}{l}
Lossy \;=\; \mathsf{let} \\
\qquad Ready \;=\; send?x \rightarrow Holding(x) \\
\qquad Holding(x) \;=\; receive!x \rightarrow Holding(x) \\
\qquad\qquad\qquad\qquad \sqcap \\
\qquad\qquad\qquad\qquad Ready \\
\quad \mathsf{within} \\
\qquad Ready
\end{array}
$$

then we may be unable to prove that our protocol provides a reliable service. This process allows an unbounded number of *send* events without a corresponding *receive*; when we hide these events for the purposes of the refinement check, the parallel combination of the protocol and media is likely to have divergences, and thus no useful failures semantics.

One way of solving this problem is to incorporate the assumption that the medium cannot lose more than N consecutive messages. In the following process, we use a parameter n to count the number of messages lost since the last successful delivery.

$$
\begin{array}{l}
Lossy(N) \;=\; \mathsf{let} \\
\qquad Ready(n) \;=\; send?x \rightarrow Holding(x, n) \\
\qquad Holding(x, n) \;=\; \text{if } n < N \text{ then} \\
\qquad\qquad\qquad\qquad\qquad receive!x \rightarrow Ready(0) \\
\qquad\qquad\qquad\qquad\qquad \sqcap \\
\qquad\qquad\qquad\qquad\qquad Ready(n+1) \\
\qquad\qquad\qquad\qquad \text{else} \\
\qquad\qquad\qquad\qquad\qquad receive!x \rightarrow Ready(0) \\
\quad \mathsf{within} \\
\qquad Ready(0)
\end{array}
$$

If our analysis requires an assumption relating a nondeterministic choice of behaviours in the supporting media to actions of the protocol components, then this behaviour should not be modelled using internal choice. Instead, we should introduce one or more events to distinguish between the alternative outcomes: these events can then be shared with another process expressing the required assumption.

Example 59 (Protocols). In the following process, which again describes a lossy medium that cannot lose more than N consecutive messages, the event *lose* is used to represent the loss of the current message:

$$
\begin{array}{l}
Lossy'(N) \;=\; \mathsf{let} \\
\qquad\qquad Ready(n) \;=\; send?x \rightarrow Holding(x, n) \\
\qquad\qquad Holding(x, n) \;=\; receive!x \rightarrow Holding(x, 0) \\
\qquad\qquad\qquad\qquad \Box \\
\qquad\qquad\qquad\qquad n < N \;\&\; lose \rightarrow Ready(n + 1) \\
\qquad \mathsf{within} \\
\qquad\qquad Ready(0)
\end{array}
$$

Note that $Lossy'(N) \setminus \{lose\} \;=\; Lossy(N)$.

If the sender is designed to retransmit any message that remains unacknowledged, then we may wish to incorporate an assumption about the timing of any retransmission. The following process, considered as a constraint, insists that no retransmission will take place until it is certain that a message has been lost.

$$
\begin{array}{l}
Timing \;=\; \mathsf{let} \\
\qquad\qquad NoTimeout \;=\; loss \rightarrow Timeout \\
\qquad\qquad Timeout \;=\; timeout \rightarrow NoTimeout \\
\qquad \mathsf{within} \\
\qquad\qquad NoTimeout
\end{array}
$$

If we add *Timing* to the process representing the combination of supporting media and protocol components, we will find that the resulting process provides a reliable service. If

$$Protocol' \;=\; Sender' \parallel Receiver'$$

$$
\begin{array}{l}
Media' \;=\; Lossy'(N)[send \leftarrow sender\_send, receive \leftarrow receiver\_receive] \\
\qquad\qquad \parallel \\
\qquad\qquad Buffer(N)[send \leftarrow receiver\_send, receive \leftarrow sender\_receive]
\end{array}
$$

$$Internal' \;=\; Internal \cup \{lose\}$$

then

$$Service \;\sqsubseteq\; (Protocol' \parallel Media' \parallel Timing) \setminus Internal'$$

8 Tools

We can use our language of processes to describe and analyse patterns of behaviour. Although this is a formal language, the descriptions that we present may be informal, in the following sense: we may present processes—or traces, or failures—to explain certain aspects of a property or design, without intending

that this explanation should be comprehensive, or even consistent. That is, the language is formal, but its usage is not. This use of process or instance models corresponds to the use of state and sequence diagrams to communicate features of a design, perhaps as a basis for discussion.

Alternatively, we may take advantage of the formal semantics, and the analysis tools, by presenting processes as complete descriptions of a component or property, at some level of abstraction. Our analysis of these processes, suitably interpreted, may then allow us to draw conclusions about the components or properties described. When we use the language in this way, we may make discoveries, or establish properties, that would not be possible in informal application: the tools amplify our modelling effort; they allow us to achieve things that would not be possible through manual labour.

In this section, we will concentrate upon the use of processes to support formal analysis, and hence upon the application of analysis tools. The following exercises require the use of *ProBE* and *FDR*: a behavioural explorer and refinement checker, respectively. Information on how to obtain these tools may be found at the website *www.usingcsp.com*, or by contacting the author of this chapter. We will present our definitions using a 'machine-readable' notation used by the tools: a simple-but-effective approximation of the language used in the previous sections.

The key features of this notation are the use of ASCII characters to approximate the CSP operators and the requirement that all events are declared as channels.

`STOP`	$\sim$	$STOP$	`P /\ Q`	$\sim$	$P \triangle Q$
`a -> P`	$\sim$	$a \rightarrow P$	`P [A || B] Q`	$\sim$	$P \llbracket A \mid B \rrbracket Q$
`P [] Q`	$\sim$	$P \Box Q$	`P |[ A ]| Q`	$\sim$	$P \llbracket A \rrbracket Q$
`P |~| Q`	$\sim$	$P \sqcap Q$	`P ||| Q`	$\sim$	$P \mid\mid\mid Q$
`P [> Q`	$\sim$	$P \triangleright Q$	`P[a <- b]`	$\sim$	$P[a \leftarrow b]$
`P ; Q`	$\sim$	$P \,;\, Q$	`P \ Q`	$\sim$	$P \setminus Q$
`SKIP`	$\sim$	**skip**			

For example, the declarations

```
DOOR = {1,2,3,4}

channel coin, tea, coffee channel open, close :~DOOR
```

introduce events used in the *Vending Machines* and *Car Doors* examples of the previous sections.

A declarative language of sets, sequences, and functions is provided for the data expressions used in process definitions; below, we have a few examples.

`a,b,c`	$\sim$	$\{a, b, c\}$	`e == f`	$\sim$	$e = f$
`union(s,t)`	$\sim$	$s \cup t$	`p and q`	$\sim$	$p \wedge q$
`inter(s,t)`	$\sim$	$s \cap t$	`p or q`	$\sim$	$p \vee q$
`diff(s,t)`	$\sim$	$s \setminus t$	`not(p)`	$\sim$	$\neg p$

A useful feature of the language is the use of `{| m, n, ...|}` to denote the set of all *productions* of the names `m`, `n`, ... : that is, all of the possibly-compound events whose names begin with these components.

A minor deficiency is the lack of any direct support for process alphabets; we cannot write simply `P || Q` or `P || Q || R`; we must provide explicit annotations. For example, the parallel combination of *Matthew*, *Marina*, and *Alan* defined in Example 30 could be described as follows.

```
let
  A(1) = {red,green}
  A(2) = {blue,green}
  A(3) = {red,blue,green}
  P(1) = Matthew
  P(2) = Marina
  P(3) = Alan
within
  || i :~{1,2,3} @ [A(i)] P(i)
```

In our use of the analysis tool *FDR*, we may make assertions about the processes that we present; the most useful of these are:

`P [T= Q`	$\sim$	$P \sqsubseteq_T Q$
`P [FD= Q`	$\sim$	$P \sqsubseteq_{FD} Q$
`P :[deadlock free]`	$\sim$	P is free from deadlock

8.1 Refinement

If we declare

```
channel coin, coffee, tea
```

then we may introduce, and analyse, a number of simple processes representing vending machines:

```
OCM = coin -> coffee -> STOP

 CM = coin -> coffee -> CM

 VM = coin ->
        (coffee -> VM
         []
         tea -> VM)
```

```
TM = coin -> tea -> TM

NVM = coin ->
        (coffee -> NVM
         |~|
         tea -> NVM)
```

(a) Use the *ProBE* tool to explore the behaviour of the processes `OCM` and `CM`. Explain the (obvious) difference between them.

(b) Which of the following assertions will be true, and why? Use the *FDR* tool to check your answers.

```
assert OCM [T= CM
assert CM [FD= OCM
assert CM [T= OCM
assert OCM [FD= CM
```

(c) The traces model provides an inadequate treatment of nondeterminism. Explain why this is the case, with reference to one or more of the following assertions:

```
assert VM [T= NVM
assert VM [FD= NVM
assert NVM [T= VM
assert NVM [FD= VM
```

(d) Which of the following assertions will be true, and why? Use the *FDR* tool to check your answers.

```
assert CM [T= VM
assert CM [FD= VM
assert VM [T= CM
assert VM [FD= CM
```

(e) Explain the difference between the processes `CM |~| TM` and `NVM`. Which of these is a Failures–Divergences refinement of the other?

(f) There are six different processes that are refinements of `NVM` in the Failures–Divergences model, including `NVM` itself. What are the others? Draw a diagram to show how these six processes are related under the $\sqsubseteq_{FD}$ refinement ordering.

8.2 Sequential Design

A machine used in radiation therapy can create two types of *beam*: a low-current beam, which plays directly upon the patient, and a high-current beam, which needs to be diffused by a beam flattener or *shield*. Using a remote terminal, an operator may select and perform a treatment; they are also able to cancel a treatment that has been selected but not yet performed.

We can produce a description of the behaviour of this machine using the following events:

```
datatype TREATMENT = high | low
channel select, current, shield : TREATMENT
channel treat, cancel
```

The event `select.t` represents the selection of treatment `t`, the event `treat` represents the performance of a treatment (the firing of the beam), and the event `cancel` represents a cancellation. The keyword `datatype` is used here to introduce a type with exactly two elements, `high` and `low`.

The machine is configured initially for low-current treatment.

```
System = SetLow ; Ready

Ready =
  select.high -> HighTreatment ; Ready
  []
  select.low -> LowTreatment ; Ready

HighTreatment =
  (SetHigh ; treat -> SetLow) /\ cancel -> SKIP

LowTreatment =
  treat -> SKIP /\ cancel -> SKIP
```

When a high-current treatment is required, the machine is configured accordingly, then re-configured for low-current use once treatment is completed. The configuration processes are nondeterministic, but neither will terminate until both current and shield have been moved to the new setting.

```
SetHigh =
  current.high -> shield.high -> SKIP
  |~|
  shield.high -> current.high -> SKIP

SetLow =
  current.low -> shield.low -> SKIP
  |~|
  shield.low -> current.low -> SKIP
```

The event `current.t` represents the current being set for treatment `t`, and the event `shield.t` represents the shield moving into the correct position for `t`.

An important consideration in the operation of this machine is that the beam and shield setting should always match when treatment is performed. If the beam is fired on low-current when the shield is high, then the patient will not be exposed to sufficient radiation; what is worse, if the beam is fired on high-current when the shield is low, then immediate, serious injury or death will occur.

Of the events introduced above, only `select`, `treat`, and `cancel` describe part of the interface presented at the remote terminal; although the `current` and `shield` events are critical to the correct behaviour of the system, their occurrence is not visible to the operator.

(a) Write a process `OperatorView` to describe the behaviour of the system from the operator's point of view. Check that your description is correct by confirming that the following refinements hold:

```
OperatorView [FD= System \ {| shield, current |}
System \ {| shield,current |} [FD= OperatorView
```

(b) Write a process `Safe` to characterize those sequences of events from the set `{| current, shield, treat |}` in which `treat` occurs only when the current and shield are at the same setting, high or low. Use this process to test whether the design of the system is safe, by checking the following refinement:

```
Safe [T= System \ {| select, cancel |}
```

(c) The trace refinement check in the previous part should have uncovered at least one problem with the design of the system. Can this problem be solved simply by eliminating the nondeterminism in the two processes `SetHigh` and `SetLow`? If not, propose your own solution, and show that it works by repeating the refinement check.

(d) An alternative approach to detecting a problematic trace is to place a monitor process in parallel with that describing the system, and arrange for this process to signal an error should such a trace be performed. Using the event defined by

```
channel error
```

together with `current`, `shield`, and `treat`, write a process `Monitor` that offers to perform `error` whenever the system performs `treat` when the beam is on the high setting and the shield is on the low setting, or vice versa. Confirm that this would be enough to detect the problem in design by performing the refinement check

```
CHAOS({| current, shield, treat, select, reset |}) [FD=
  System [| {| current, shield, treat |} |] Monitor
```

Where `CHAOS(A)` is a process (pre-defined in *FDR*) that can perform or refuse any event from set `A`, but does not diverge (unlike the process *CHAOS* in our original language, which takes no parameter, and may diverge immediately).

(e) In this exercise, we used a process to characterize all of the good traces of a system, with respect to a particular property, and a particular set of events. Explain why it is impossible for us to use a process to describe directly all of the bad traces of a system.

8.3 Parallel Combination

The behaviour of a particular design of central locking system can be described using three kinds of process: one representing a door lock, another representing a door button (or lever), and another representing the central control module. For the purposes of this exercise, we will assume that communication between the various components is immediate and reliable.

We will introduce sets of door numbers, command values, and movements made by door buttons (or levers):

```
DOORS = {1,2}
datatype COMMANDS = lock | unlock
datatype MOVES = up | down
```

and events to represent: the action of issuing a command to the controller via the remote control; the transmission of a command to a door lock (either from the controller or from the button); the movement of a door button; and the opening and closing of doors.

```
channel remote :~COMMANDS
channel command :~DOORS . COMMANDS
channel button :~DOORS . MOVES
channel open, close :~DOORS
```

An additional event `crash` will to used to represent the controller module deciding, or being informed, that a collision is imminent (or even occurring).

```
channel crash
```

(a) A door should open and close only if unlocked. It can be locked only if closed: it will ignore lock and unlock commands if open. Write a process vDoor(d) to describe the behaviour of door `d`, using events from the set

```
{| open.d, close.d, command.d |}
```

(b) A door button can move up and down alternately. When it is moved down, it sends a lock command to that door; when it is moved up, it sends an unlock command. Write a process `Button(d)` to describe the behaviour of the button on door `d`, using events from the set

```
{| button.d, command.d |}
```

(c) The controller will send a lock command to each of the two doors in response to the signal `remote.lock`, and an unlock command in response to vremote.unlock. It will also send unlock commands in response to the `crash` signal. Write a process *Controller* to describe the behaviour of the controller, using events from the set

```
{| remote, command, crash |}
```

(d) The controller and the buttons send commands to the door lock independently of each other. Using parallel combination and partial interleaving, write a process `System` to describe the behaviour of the central locking system with two doors, two buttons, and a controller component.

(e) A key requirement upon the design is that the doors should be capable of being opened immediately after a crash. Write a process `Escape` to express this property, using events from the set

```
{| crash, open, close |}
```

Would it be useful to check the following refinement?

```
Escape [FD= System \ {| command, button, remote |}
```

(f) Instead of hiding the button and remote events, we may ignore them. We can do this by writing a new property process in which, at every stage, these

events may or may not be available, and it would make no difference if they occurred. Write a new process `NewEscape`, using events from the set

```
{|crash, open, close, button, remote |}
```

so that the refinement check

```
NewEscape [FD= System \ {| command |}
```

reveals further, useful information about the design.

(g) The refinement check in the previous part will not succeed. It is possible, however, to show that our design will achieve the objective of permitting escape under the assumption that no buttons are pressed after the `crash` event occurs. Write a process `Assume` using events from the set

```
{| crash, button |}
```

so that

```
NewEscape [FD=
  (System [| {| crash, button |} |] Assume) \ {| command |}
```

(h) Alternatively, we might change our description of the system so that the problematic behaviours do not arise. Depending upon the circumstances, such a change may reflect either an improved design, or an improved description of an already adequate design.
Here, we may choose to model a mechanism by which the controller is informed whenever a door is locked or unlocked, using events

```
channel report :~DOORS . COMMANDS
```

Write a process `NewSystem` to describe the behaviour of the new design, revising the definitions of the component processes, and the parallel combinations, as necessary, so that

```
NewEscape [FD=
  NewSystem \ {| command, remote |}
```

9 Further Reading

There are two publications on CSP that are particularly useful:

- C. A. R. Hoare. *Communicating Sequential Processes.*
- A. W. Roscoe. *Theory and Practice of Concurrency.*

Both of these are available for download: links to the latest versions can be found at *www.usingcsp.com*, together with information about tools for exploring behaviour and checking refinements.

Developing and Reasoning About Probabilistic Programs in *pGCL*

Annabelle McIver[1] and Carroll Morgan[2]

[1] Department of Computer Science
Macquarie University
NSW, Australia
[2] Department of Computer Science and Engineering
University of New South Wales
NSW, Australia

As explained in Chapter 1, Dijkstra's guarded-command language, which we call *GCL*, was introduced as an intellectual framework for rigorous reasoning about imperative sequential programs; one of its novelties was that it contained explicit "demonic" nondeterminism, representing abstraction from (or ignorance of) which of two program fragments will be executed. By introducing *probabilistic* nondeterminism into *GCL*, we provide a means with which also probabilistic programs can be rigorously developed and reasoned about.

The programming logic of "weakest preconditions" for *GCL* becomes a logic of "greatest pre-expectations" for what we call *pGCL*. An *expectation* is a generalized predicate suitable for expressing quantitative properties such as "the probability of achieving a postcondition".

pGCL is suitable for describing random algorithms, at least over discrete distributions. In our presentation of it and its logic we give a number of small examples, and two case studies. The first illustrates probabilistic "almost-certain" termination; the second case study illustrates approximated probabilities, abstraction and refinement.

After a brief historical account of work on probabilistic semantics in Section 1, Section 2 gives a brief and shallow overview of *pGCL*, somewhat informal and concentrating on simple examples. Section 3 sets out the definitions and properties of *pGCL* systematically, and Section 4 treats an example of reasoning about probabilistic loops, showing how to use probabilistic *invariants*. Section 5 illustrates termination arguments via probabilistic *variants* with a thorough treatment of Rabin's choice-coordination algorithm [219]; Section 6 illustrates abstraction and refinement, as well as "approximated probabilities", by giving a two-level treatment of an almost-uniform selection algorithm. An impression of *pGCL* can be gained by reading Sections 2 and 4, with finally a glance over Sections 3.1 and 3.2; more thoroughly one would read Sections 2, 3.1 and 3.2, then 2 (again) and finally 4. The more theoretical Section 3.3 can be skipped on first reading. Appendix A describes basic concepts of probability theory needed in this chapter.

A. Cavalcanti, A. Sampaio, and J. Woodcock (Eds.): PSSE 2004, LNCS 3167, pp. 123–155, 2006.

1 Introduction

Probabilistic programs and systems are increasingly relevant: often random algorithms are computationally feasible where their deterministic counterparts are not; some concurrent applications are impossible without the symmetry breaking that randomisation provides; and in hybrid systems the low-level hardware might be represented by probabilistic program text that models quantitative unreliability. Because of that relevance, there has been a renewed interest in techniques for establishing the correctness of such programs—for the more widespread they become, the more we will depend on understanding their behaviour, and their limits, exactly.

In this tutorial chapter we address that last concern, of understanding: we survey a method for rigorous reasoning about probabilistic programs and systems. We give an impression of how they work, an *operational* view, and we suggest how we should reason about them, a *logical* view—and we show how the two views are designed to fit together.

We use Dijkstra's Guarded Command Language *GCL* [81] as a simple and "pared-down" syntax for presenting our ideas: it is a weakest-precondition based method of describing computations and their meaning; here we extend it to probabilistic programs, and we give examples of its use.

Most sequential programming languages contain a construct for "deterministic" choice, where the program makes a selection in a predictable way: for example, in

$$\textbf{if } \mathit{test} \textbf{ then } \mathsf{This} \textbf{ else } \mathsf{That} \textbf{ fi} \tag{1}$$

the two-way choice between This and That is determined by *test* and the current state.

In contrast, Dijkstra's language of guarded commands brings to prominence nondeterministic or "demonic" choice, in which the program's behaviour is *not* predictable, is not determined by the current state. At first [81], demonic choice was presented as a consequence of "overlapping guards", as almost an accident, but as its importance became more widely recognized it developed a life of its own. Nowadays it merits an explicit operator: the construct

$$\mathsf{This} \sqcap \mathsf{That}$$

chooses between the alternatives unpredictably and, as a specification, indicates abstraction from the issue of which will be executed. The customer will be happy with either This or That; and the implementor may choose between them according to his own concerns. An alternative but equivalent view is that the choice between the alternatives is made at runtime by an adversarial "demon" whose aim is to make the program as unlikely as possible to achieve its goal.

Early research on probabilistic semantics took a different route: demonic choice was not regarded as fundamental. Rather it was abandoned altogether, being replaced by probabilistic choice [140, 90, 89, 133, 132], written for example

$$\mathsf{This} \ {}_{p}\!\oplus \ \mathsf{That}$$

to indicate a program that behaved like This with probability p, but otherwise like That. Without demonic choice, however, probabilistic semantics was divorced from the contemporaneous work on specification and refinement: there was no longer any means of abstraction.

More recently it has been discovered [131, 197, 247] how to bring the two topics back together, taking the more natural approach of *adding* probabilistic choice, while retaining demonic choice. In fact deterministic choice is a special case of probabilistic choice, which in turn is a refinement of demonic choice.

We give the resulting probabilistic extension of *GCL* the name "*pGCL*".

2 An Impression of *pGCL*

Let square brackets $[\cdot]$ be used to embed Boolean-valued predicates within arithmetic formulae which, for reasons explained below, we call *expectations*; we allow them to range over the unit interval $[0, 1]$. Stipulating that [**false**] is 0 and [**true**] is 1 makes $[P]$ in a trivial sense the probability that a given predicate P holds: if false, P holds with probability 0; if true, it holds with probability 1.

For (our first) example, consider the simple program

$$x{:=}\,-y \quad {}_{\frac{1}{3}}\!\oplus \quad x{:=}\,+y \tag{2}$$

over variables $x, y{:}\,\mathbb{Z}$, using a construct ${}_{\frac{1}{3}}\!\oplus$ which, as explained above, we interpret as "choose the left branch $x{:=}\,-y$ with probability $1/3$, and choose the right branch with probability $1 - 1/3$".

Recall [81] that for any predicate P over *final* states, and a standard command S, the "weakest precondition" predicate $wp.\mathsf{S}.P$ acts over *initial* states: it holds just in those initial states from which S is guaranteed to reach P. (Throughout this chapter, we use *standard* to mean "non-probabilistic".) We also write $f.x$ instead of $f(x)$ for function f applied to argument x, with left association. Now suppose S is probabilistic, as Program (2) is; what can we say about the *probability* that $wp.\mathsf{S}.P$ holds in some initial state?

It turns out that the answer is just $wp.\mathsf{S}.[P]$, once we generalize $wp.\mathsf{S}$ to expectations instead of predicates. For that, we begin with the two definitions

$$wp.(x{:=}\,E).R \;\widehat{=}\; \text{"}R \text{ with } x \text{ replaced everywhere by } E\text{"} \,[1] \tag{3}$$

$$wp.(\mathsf{S}\ {}_{p}\!\oplus \mathsf{T}).R \;\widehat{=}\; p * wp.\mathsf{S}.R + (1{-}p) * wp.\mathsf{T}.R \tag{4}$$

in which R is an expectation, and for our example program we ask

> what is the probability that the predicate "the final state will satisfy $x \geq 0$" holds in some given initial state of the Program (2)?

[1] In the usual way, we take account of free and bound variables, and if necessary rename to avoid variable capture.

To find out, we calculate $wp.\mathsf{S}.[P]$ in this case; that is

$$\begin{array}{lll} & wp.(x:= -y \;{}_{1/3}\!\oplus\; x:= +y).[x \geq 0] & \\ \equiv & \quad (1/3) * wp.(x:= -y).[x \geq 0] & \text{using (3)} \\ & + (2/3) * wp.(x:= +y).[x \geq 0] & \\ \equiv & (1/3)\,[-y \geq 0] + (2/3)\,[+y \geq 0] & \text{using (3)} \\ \equiv & [y < 0]\,/3 \;+\; [y = 0] \;+\; 2\,[y > 0]\,/3 & \text{using arithmetic} \end{array}$$

Thus our answer is the last arithmetic formula above, which we could call a "pre-expectation"—and the probability we seek is found by reading off the formula's value for various initial values of y, getting

$$\begin{array}{lll} \text{when } y < 0, & 1/3 + 0 + 2(0)/3 & = \;\; 1/3 \\ \text{when } y = 0, & 0/3 + 1 + 2(0)/3 & = \;\; 1 \\ \text{when } y > 0, & 0/3 + 0 + 2(1)/3 & = \;\; 2/3 \end{array}$$

Those results indeed correspond with our operational intuition about the effect of ${}_{1/3}\!\oplus$. Later we explain the use of "$\equiv$" rather than "$=$".

The above remarkable generalisation of sequential program correctness is due to Kozen [140], but in its original form was restricted to programs that did not contain demonic choice $\sqcap$. When He *et al.* [131] and Morgan *et al.* [197] successfully added demonic choice, it became possible to begin the long-overdue integration of probabilistic programming and formal program development: in the latter, demonic choice—as *abstraction*—plays a crucial role in specifications. The extension was based on a general approach to probabilistic power-domains due to Jones and Plotkin [132, 133], which recently has been further developed by Tix *et al.* [247].

To illustrate the use of abstraction, in our second example we abstract from probabilities: a demonic version of Program (2) is much more realistic in that we set its probabilistic parameters only within some tolerance. We say informally (but still with precision) that

$$\left.\begin{array}{l} - \;\; x:= -y \text{ is to be executed with probability } \textit{at least } 1/3, \\ - \;\; x:= +y \text{ is to be executed with probability } \textit{at least } 1/4 \text{ and} \\ - \;\; \text{it is certain that one or the other will be executed.} \end{array}\right\} \quad (5)$$

Equivalently we could say that alternative $x:= -y$ is executed with probability between $1/3$ and $3/4$, and that otherwise $x:= +y$ is executed (therefore with probability between $1/4$ and $2/3$).

With demonic choice we can write Specification (5) as

$$\begin{array}{rl} & x:= -y \;{}_{1/3}\!\oplus\; x:= +y \\ \sqcap \;\; & x:= -y \;{}_{3/4}\!\oplus\; x:= +y \end{array} \quad (6)$$

because we do not know or care whether the left or right alternative of $\sqcap$ is taken—and it may even vary from run to run of the program, resulting in an

"effective" ${}_p\oplus$ with p somewhere between the two extremes. A convenient notation for (6) would be based on the abbreviation

$$S \;{}_p\oplus_q\; T \;\;\widehat{=}\;\; (S \;{}_p\oplus\; T) \sqcap (T \;{}_q\oplus\; S) \quad \text{for } p + q \leq 1$$

we would then write $x{:=}\,-y \;{}_{\frac{1}{3}}\oplus_{\frac{1}{4}}\; x{:=}\,+y$.

To treat Program (6), we define the command

$$wp.(S \sqcap T).R \;\;\widehat{=}\;\; wp.S.R \;\mathit{min}\; wp.T.R \tag{7}$$

using *min* because we regard demonic behaviour as attempting to make the achieving of R as *im*-probable as it can. Repeating our earlier calculation (but more briefly) gives this time

$$\begin{array}{lll} & wp.(\text{ Program (6) }).[x \geq 0] & \\ \equiv & \quad [y \leq 0]\,/3 + 2\,[y \geq 0]\,/3 & \text{using (3), (3), (7)} \\ \mathit{min} & \quad 3\,[y \leq 0]\,/4 + [y \geq 0]\,/4 & \\ \equiv & [y < 0]\,/3 \;+\; [y = 0] \;+\; [y > 0]\,/4 & \text{using arithmetic} \end{array}$$

Our interpretation is now

- When y is initially negative, the demon chooses the left branch of $\sqcap$ because that branch is more likely (*2/3 vs. 1/4*) to execute $x{:=}\,+y$—the best we can say then is that $x \geq 0$ will hold with probability at least *1/3*.
- When y is initially zero, the demon cannot avoid $x \geq 0$—either way the probability of $x \geq 0$ finally is 1.
- When y is initially positive, the demon chooses the right branch because that branch is more likely to execute $x{:=}\,-y$—the best we can say then is that $x \geq 0$ finally with probability at least *1/4*.

The same interpretation holds if we regard $\sqcap$ as abstraction. Suppose Program (6) represents some mass-produced physical device and, by examining the production method, we have determined the tolerance (5) on the devices produced. If we were to buy one arbitrarily, all we could conclude about its probability of establishing $x \geq 0$ is just as calculated above.

Refinement is the converse of abstraction: for two commands S, T we define

$$S \sqsubseteq T \;\;\widehat{=}\;\; wp.S.R \Rrightarrow wp.T.R \quad \text{for all } R \tag{8}$$

where we write $\Rrightarrow$ for "everywhere no more than" (which ensures $[\mathbf{false}] \Rrightarrow [\mathbf{true}]$ as the notation suggests). From (8) we see that in the special case when R is an embedded predicate $[P]$, the meaning of $\Rrightarrow$ ensures that a refinement T of S is at least as likely to establish P as S is. That accords with the usual definition of refinement for standard programs—for then we know $wp.S.[P]$ is either 0 or 1, and whenever S is certain to establish P (whenever $wp.S.[P] \equiv 1$) we know that T also is certain to do so (because then $1 \Rrightarrow wp.T.[P]$).

For our third example we prove a refinement: consider the program

$$x{:=}\,-y \;\;{}_{\frac{1}{2}}\oplus\;\; x{:=}\,+y \tag{9}$$

which clearly satisfies Specification (5); thus it should refine Program (6). With Definition (8), we find for any R that

$$
\begin{array}{ll}
& wp.(\text{ Program (9) }).R \\
\equiv & wp.(x{:=}\,-y).R/2 \;\;+\;\; wp.(x{:=}\,+y).R/2 \\
\equiv & R^-/2 + R^+/2 \qquad\qquad \text{introduce abbreviations} \\
\equiv & \quad (3/5)(R^-/3 + 2R^+/3) \qquad\qquad \text{arithmetic} \\
& +\;\; (2/5)(3R^-/4 + R^+/4) \\
\Leftarrow & \quad R^-/3 + 2R^+/3 \qquad\qquad \text{any linear combination exceeds } \mathit{min} \\
& \mathit{min}\;\; 3R^-/4 + R^+/4 \\
\equiv & wp.(\text{ Program (6) }).R
\end{array}
$$

The refinement relation (8) is indeed established for the two programs.

The introduction of $3/5$ and $2/5$ in the third step can be understood by noting that demonic choice $\sqcap$ can be implemented by any probabilistic choice whatever: in this case we used ${}_{\frac{3}{5}}\!\oplus$. Thus a proof of refinement at the program level might read

$$
\begin{array}{ll}
& \text{Program (9)} \\
= & x{:=}\,-y \;\;{}_{\frac{1}{2}}\!\oplus\;\; x{:=}\,+y \\
= & \quad (x{:=}\,-y \;\;{}_{\frac{1}{3}}\!\oplus\;\; x{:=}\,+y) \qquad\qquad \text{arithmetic} \\
& {}_{\frac{3}{5}}\!\oplus\;\; (x{:=}\,-y \;\;{}_{\frac{3}{4}}\!\oplus\;\; x{:=}\,+y) \\
\sqsupseteq & \quad x{:=}\,-y \;\;{}_{\frac{1}{3}}\!\oplus\;\; x{:=}\,+y \qquad\qquad (\sqcap) \sqsubseteq ({}_p\!\oplus) \text{ for any } p \\
& \sqcap\;\; x{:=}\,-y \;\;{}_{\frac{3}{4}}\!\oplus\;\; x{:=}\,+y \\
\equiv & \text{Program (6)}
\end{array}
$$

3 Presentation of Probabilistic *GCL*

In this section we give a concise presentation of probabilistic *GCL*—*pGCL*: its definitions, how they are to be interpreted and their (healthiness) properties.

3.1 Definitions of *pGCL* Commands

In *pGCL*, commands act between "expectations" rather than predicates, where an *expectation* is an expression over (program or state) variables that takes its value in the unit interval $[0, 1]$. (A more general treatment is possible in which expectations are arbitrarily non-negative but bounded [197, 181].) To retain the use of predicates, we allow expectations of the form $[P]$ when P is Boolean-valued, defining [**false**] to be 0 and [**true**] to be 1.

Implication-like relations between expectations are

$$
\begin{array}{rcl}
R \Rrightarrow R' & \widehat{=} & R \text{ is everywhere no more than } R' \\
R \equiv R' & \widehat{=} & R \text{ is everywhere equal to } R' \\
R \Lleftarrow R' & \widehat{=} & R \text{ is everywhere no less than } R'
\end{array}
$$

The probabilistic guarded command language *pGCL* acts over "expectations" rather than predicates: *expectations* take values in $[0, 1]$.

$wp.(x := E).R$	The expectation obtained after replacing all free occurrences of x in R by E, renaming bound variables in R if necessary to avoid capture of free variables in E.
$wp.\mathbf{skip}.R$	R
$wp.(\mathsf{S};\ \mathsf{T}).R$	$wp.\mathsf{S}.(wp.\mathsf{T}.R)$
$wp.(\mathsf{S} \sqcap \mathsf{T}).R$	$wp.\mathsf{S}.R\ \mathit{min}\ wp.\mathsf{T}.R$
$wp.(\mathsf{S}\ {}_p\!\oplus\ \mathsf{T}).R$	$p * wp.\mathsf{S}.R + (1-p) * wp.\mathsf{T}.R$
$\mathsf{S} \sqsubseteq \mathsf{T}$	$wp.\mathsf{S}.R \Rrightarrow wp.\mathsf{T}.R$ for all R

- R is an expectation (possibly but not necessarily $[P]$ for a predicate P);
- P is a predicate (not an expectation);
- $*$ is multiplication;
- S, T are probabilistic guarded commands (inductively);
- p is an expression over the program variables (possibly but not necessarily a constant), taking a value in $[0, 1]$; and
- x is a variable (or a vector of variables).

Deterministic choice **if** B **then** S **else** T **fi** is a special case of probabilistic choice: it is just $\mathsf{S}\ {}_{[B]}\!\oplus\ \mathsf{T}$. Recursions are handled by least fixed points in the usual way; in practice however, the special case of loops is more easily treated using (probabilistic) invariants and variants.

Fig. 1. *pGCL*—the probabilistic Guarded Command Language

Note that $\models P \Rightarrow P'$ exactly when $[P] \Rrightarrow [P']$, and so on; that is the motivation for the symbols chosen.

The definitions of the commands in *pGCL* are given in Fig. 1.

3.2 Interpretation of *pGCL* Expectations

In its full generality, an expectation is a function describing how much each program state "is worth".

The special case of an embedded predicate $[P]$ assigns to each state a worth of 0 or of 1: states satisfying P are worth 1, and states not satisfying P are worth 0. The more general expectations arise when one estimates, in the *initial* state of a probabilistic program, what the worth of its *final* state will be. That

estimate, the "expected worth" of the final state, is obtained by summing over all final states

> the worth of the final state multiplied by the probability the program "will go there" from the initial state.

Naturally the "will go there" probabilities depend on "from where", and so that expected worth is a function of the initial state.

When the worth of final states is given by $[P]$, the expected worth of the initial state turns out to be just the probability that the program will reach P. That is because

$$
\begin{array}{ll}
& \text{expected worth of initial state} \\
\equiv & \quad (\text{probability } \mathcal{S} \text{ reaches } P) * (\text{worth of states satisfying } P) \\
+ & (\text{probability } \mathcal{S} \text{ does not reach } P) * (\text{worth of states not satisfying } P) \\
\equiv & \quad (\text{probability } \mathcal{S} \text{ reaches } P) * 1 \\
+ & (\text{probability } \mathcal{S} \text{ does not reach } P) * 0 \\
\equiv & \text{probability } \mathcal{S} \text{ reaches } P
\end{array}
$$

where, of course, matters are greatly simplified by the fact that all states satisfying P have the same worth. We must, however, moderate this to "the greatest guaranteed probability" when there is demonic choice: this is why the general judgement is the inequality $p \Rrightarrow wp.\mathcal{S}.[P]$ rather than the special case of equality given at (10).

Typical analyses of programs $\mathcal{S}$ in practice lead to conclusions of the form

$$p \;\equiv\; wp.\mathcal{S}.[P] \tag{10}$$

for some p and P which, given the above, we can interpret in two equivalent ways:

1. the expected worth $[P]$ of the final state is at least the value of p in the initial state; or
2. the probability that $\mathcal{S}$ will establish P is at least p.

Each interpretation is useful, and in the following example we can see them acting together: we ask for the probability that two fair coins when flipped will show the same face, and calculate

$$
\begin{array}{ll}
& wp.\begin{pmatrix} c{:=}\ H \ {}_{\frac{1}{2}}\!\oplus\ c{:=}\ T\,; \\ d{:=}\ H \ {}_{\frac{1}{2}}\!\oplus\ d{:=}\ T \end{pmatrix}.[c=d] \\
\equiv & \qquad\qquad {}_{\frac{1}{2}}\!\oplus, :=\text{ and sequential composition} \\
& wp.(c{:=}\ H \ {}_{\frac{1}{2}}\!\oplus\ c{:=}\ T).([c=H]\,/2 + [c=T]\,/2) \\
\equiv & \quad (1/2)([H=H]\,/2 + [H=T]\,/2) \qquad\qquad {}_{\frac{1}{2}}\!\oplus \text{ and } := \\
+ & (1/2)([T=H]\,/2 + [T=T]\,/2) \\
\equiv & (1/2)(1/2 + 0/2) + (1/2)(0/2 + 1/2) \qquad\qquad \text{definition } [\cdot] \\
\equiv & \ 1/2 \qquad\qquad \text{arithmetic}
\end{array}
$$

We can then use the second interpretation above to conclude that the faces are the same with probability (at least) $1/2$. Knowing there is no demonic choice in the program, we can in fact say it is exact.

But part of the above calculation involves the more general expression

$$wp.(c{:=}\ H\ {}_{\frac{1}{2}}\!\oplus c{:=}\ T).([c = H]/2 + [c = T]/2)$$

and what does that mean on its own? It must be given the first interpretation, since its post-expectation is not of the form $[P]$, and it means

> the expected value of the expression $[c = H]/2 + [c = T]/2$ after executing $c{:=}\ H\ {}_{\frac{1}{2}}\!\oplus c{:=}\ T$,

which the calculation goes on to show is in fact $1/2$. But for our overall conclusions we do not need to think about the intermediate expressions—they are only the "glue" that holds the overall reasoning together.

Exercise 1. We consider again the two coin-like variables c and d which are flipped in various ways. We use the notation $c{:=}\ H\ {}_p\!\oplus\ T$ to represent the assignment of H to c with probability p, and of T with probability $1-p$; similarly, we write $d{:=}\ H\ {}_p\!\oplus\ T$.

1. What if one of the two coins is not fair? Calculate

$$\begin{array}{ll} & wp.(c{:=}\ H\ {}_p\!\oplus\ T;\ \ d{:=}\ H\ {}_{1/2}\!\oplus\ T).\,[c = d] \\ \text{and} & wp.(c{:=}\ H\ {}_{1/2}\!\oplus\ T;\ \ d{:=}\ H\ {}_q\!\oplus\ T).\,[c = d] \end{array}$$

2. What if one of the two coins is not even flipped, but rather is placed face-up or -down at will? (At *whose* will?) Calculate

$$\begin{array}{ll} & wp.(c{:=}\ H \sqcap T;\ \ d{:=}\ H\ {}_{1/2}\!\oplus\ T).\,[c = d] \\ \text{and} & wp.(c{:=}\ H\ {}_{1/2}\!\oplus\ T;\ \ d{:=}\ H \sqcap T).\,[c = d]\ . \end{array}$$

3. Of the five answers to the questions above, (including the two-fair-coins example in the text) one is conspicuous. Which one? How do you explain that answer?

3.3 Properties of *pGCL*

Recall that all *GCL* constructs satisfy the property of conjunctivity—that is, for any *GCL* command S and post-conditions P, P' we have

$$wp.\mathsf{S}.(P \wedge P') \ \ = \ \ wp.\mathsf{S}.P \wedge wp.\mathsf{S}.P'$$

That "healthiness property" [81] is used to prove general properties of programs.

In *pGCL* the healthiness condition becomes "sublinearity" [197], a generalisation of conjunctivity:

Definition 1 (Sub-linearity). *Let a, b and c be non-negative finite reals, and R and R' expectations; then all pGCL constructs S satisfy*

$$wp.\mathsf{S}.(aR + bR' \ominus c) \quad \Leftarrow \quad a(wp.\mathsf{S}.R) + b(wp.\mathsf{S}.R') \ominus c \tag{11}$$

This property of S is called sublinearity*. We have written aR for $a * R$, and so on. Truncated subtraction $\ominus$ is defined*

$$x \ominus y \quad \widehat{=} \quad (x - y) \max 0$$

with syntactic precedence lower than $+$.

Sublinearity characterizes probabilistic *and demonic* commands. In Kozen's original probability-only formulation [140] the commands are not demonic, and there they satisfy the much stronger property of "linearity" [179].

Although it has a strange appearance, from sublinearity we can extract a number of very useful consequences, as we now show [197]. We begin with monotonicity, feasibility and scaling.

monotonicity: increasing a post-expectation can only increase the pre-expectation. Suppose $R \Rrightarrow R'$ for two expectations R, R'; then

$$\begin{array}{lll}
& wp.\mathsf{S}.R' & \\
\equiv & wp.\mathsf{S}.(R + (R' - R)) & \\
\Leftarrow & wp.\mathsf{S}.R + wp.\mathsf{S}.(R'{-}R) & \text{sublinearity with } a, b, c := 1, 1, 0 \\
\Leftarrow & wp.\mathsf{S}.R & R'{-}R \text{ well defined, hence } 0 \Rrightarrow wp.\mathsf{S}.(R'{-}R)
\end{array}$$

feasibility: pre-expectations cannot be "too large". First note that $wp.\mathsf{S}.0$ must be 0, as we show below.

$$\begin{array}{lll}
& wp.\mathsf{S}.0 & \\
\equiv & wp.\mathsf{S}.(2 * 0) & \\
\Leftarrow & 2 * wp.\mathsf{S}.0 & \text{sublinearity with } a, b, c := 2, 0, 0
\end{array}$$

Now write $\max R$ for the maximum of R over all its variables' values; then

$$\begin{array}{lll}
& 0 & \\
\equiv & wp.\mathsf{S}.0 & \text{feasibility above} \\
\equiv & wp.\mathsf{S}.(R \ominus \max R) & R \ominus \max R \equiv 0 \\
\Leftarrow & wp.\mathsf{S}.R \ominus \max R & \text{sublinearity with } a, b, c := 1, 0, \max R
\end{array}$$

But from $0 \Leftarrow wp.\mathsf{S}.R \ominus (\max R)$ we have trivially that

$$wp.\mathsf{S}.R \quad \Rrightarrow \quad \max R \tag{12}$$

which we identify as the *feasibility* condition for *pGCL*. Conveniently, the general (12) implies the earlier special case $wp.\mathsf{S}.0 \equiv 0$.

scaling: multiplication by a non-negative constant distributes through commands. Note first that $wp.\mathcal{S}.(aR) \Leftarrow a(wp.\mathcal{S}.R)$ directly from sublinearity. For $\Rrightarrow$ we have two cases: when a is 0, trivially from feasibility

$$wp.\mathcal{S}.(0 * R) \;\equiv\; wp.\mathcal{S}.0 \;\equiv\; 0 \;\equiv\; 0 * wp.\mathcal{S}.R$$

and for the other case $a \neq 0$ we reason as follows, establishing the identity $wp.\mathcal{S}.(aR) \equiv a(wp.\mathcal{S}.R)$ generally.

$$\begin{array}{lll} & wp.\mathcal{S}.(aR) & \\ \equiv & a(1/a)wp.\mathcal{S}.(aR) & a \neq 0 \\ \Rrightarrow & a(wp.\mathcal{S}.((1/a)aR)) & \text{sublinearity using } 1/a \\ \equiv & a(wp.\mathcal{S}.R) & \end{array}$$

That completes monotonicity, feasibility and scaling.

The remaining property we examine is probabilistic conjunction. Since standard conjunction $\wedge$ is not defined over numbers, we have many choices for a probabilistic analogue & of it, requiring only, for consistency with embedded Booleans, that

$$\begin{array}{l} 0 \;\&\; 0 = 0 \\ 0 \;\&\; 1 = 0 \\ 1 \;\&\; 0 = 0 \\ 1 \;\&\; 1 = 1 \end{array} \tag{13}$$

Obvious possibilities for & are multiplication $*$ and minimum min, and each of those has its uses; but neither satisfies anything like a generalisation of conjunctivity. Instead we define

$$R \;\&\; R' \;\;\widehat{=}\;\; R + R' \ominus 1 \tag{14}$$

whose right-hand side is inspired by sublinearity when $a, b, c := 1, 1, 1$. We now state a (sub-)distribution property for it, a direct consequence of sublinearity. This same operator (and its other propositional companions) was introduced by Łukasiewicz in the 1920's [103]; here we have synthesized it by quite different means.

sub-conjunctivity: the operator & subdistributes through commands. From sublinearity with $a, b, c := 1, 1, 1$ we have

$$wp.\mathcal{S}.(R \;\&\; R') \;\Leftarrow\; wp.\mathcal{S}.R \;\&\; wp.\mathcal{S}.R'$$

for all $\mathcal{S}$.

Unfortunately there does not seem to be a full (rather than sub-)conjunctivity property.

Beyond sub-conjunctivity, we say that & generalizes conjunction for several other reasons. The first is of course that it satisfies the standard properties (13).

The second reason is that sub-conjunctivity implies "full" conjunctivity for standard programs. Standard programs, containing no probabilistic choices, take

standard $[P]$-style post-expectations to standard pre-expectations: they are the embedding of *GCL* in *pGCL*, and for standard S we now show that

$$wp.\mathsf{S}.([P] \,\&\, [P']) \;\equiv\; wp.\mathsf{S}.[P] \;\&\; wp.\mathsf{S}.[P'] \tag{15}$$

First note that "$\Leftarrow$" comes directly from sub-conjunctivity above, taking R, R' to be $[P], [P']$.

For "$\Rightarrow$" we appeal to monotonicity, because $[P] \,\&\, [P'] \Rrightarrow [P]$ whence we have $wp.\mathsf{S}.([P] \,\&\, [P']) \Rrightarrow wp.\mathsf{S}.[P]$, and similarly for P'. Putting those together gives

$$wp.\mathsf{S}.([P] \,\&\, [P']) \;\Rrightarrow\; wp.\mathsf{S}.[P] \;\mathit{min}\; wp.\mathsf{S}.[P']$$

by elementary arithmetic properties of $\Rrightarrow$. But on standard expectations—which $wp.\mathsf{S}.[P]$ and $wp.\mathsf{S}.[P']$ are, because S is standard—the operators *min* and & agree.

A last attribute linking & to $\wedge$ comes straight from elementary probability theory. Let A and B be two events, unrelated by $\subseteq$ and not necessarily independent: then we can show that

> if the probabilities of A and B are at least p and q respectively, then the most that can be said about the joint event $A \cap B$ is that it has probability at least $p \,\&\, q$ [235].

The & operator also plays a crucial role in the proof [193, 181] (not given here) of the probabilistic loop rule presented and used in the next section.

Exercise 2. Say that a probabilistic program is *standard* if it takes *0/1*-valued post-expectations to *0/1*-valued pre-expectations; typical examples are programs written in *pGCL* that nevertheless do not use ${}_p\oplus$. Show that such programs distribute *minimum* for all post-expectations. For hints, consult the reference text on *pGCL* [181].

4 Probabilistic Invariants for Loops

To show *pGCL* in action, we state a proof rule for probabilistic loops and apply it to a simple example.

Just as for standard loops, we can deal with invariants and termination separately: common sense suggests that the probabilistic reasoning should be an extension of standard reasoning, and indeed that is the case. One proves a predicate invariant under execution of a loop's body; and one finds a variant that ensures the loop's eventual termination: the conclusion is that if the invariant holds initially then the invariant and the negation of the loop guard together hold finally. Probability does lead to differences, however—and here are some of them:

– The invariant *may* be probabilistic, in which case its operational meaning is more general than just "the computation remains within a certain set of states".

– The variant might *have* to be probabilistically interpreted, since the usual "must strictly decrease and is bounded below" technique is no longer adequate, even for simple cases. (It remains sound.)
– When both the invariant and the termination condition are probabilistic, one cannot use Boolean conjunction to combine "correct if terminates" and "it does terminate".

4.1 Probabilistic Invariants

In a standard loop, the invariant holds at every iteration of the loop. It describes a set of states from which continuing to execute the loop body is guaranteed to establish the postcondition, if the guard ever becomes false—that is, if termination occurs.

For a probabilistic loop we have a post-expectation rather than a postcondition, but otherwise the situation is much the same. Moreover, if that post-expectation is some $[P]$ say, then—as an aid to the intuition—we can look for an invariant that gives a lower bound on the probability that we will establish P by (continuing to) execute the loop body. Often that invariant will have the form

$$p \;*\; [I] \tag{16}$$

with p a probability and I a predicate, both expressions over the state. From the definition of $[\cdot]$ we know that the interpretation of (16) is

> probability p if I holds, and probability 0 otherwise.

We see an example of such invariants in Section 4.3.

4.2 Termination

The probability that a program will terminate generalizes the usual definition: recalling that $[\mathbf{true}] \equiv 1$ we see that a program's probability of termination is given by

$$wp.\mathsf{S}.1 \tag{17}$$

As a simple example of that, suppose S is the recursive program

$$\mathsf{S} \;\widehat{=}\; \mathsf{S} \,{}_p\!\oplus \mathbf{skip} \tag{18}$$

in which we assume that p is some constant strictly less than 1: on each recursive call, P has probability $1-p$ of termination, continuing otherwise with further recursion. Elementary probability theory shows that S terminates with probability 1 (after an expected $p/(1-p)$ recursive calls). By calculation based on (17) we see that

$$\begin{aligned} & wp.\mathsf{S}.1 \\ \equiv\; & p * (wp.\mathsf{S}.1) + (1-p) * (wp.\mathbf{skip}.1) \\ \equiv\; & p * (wp.\mathsf{S}.1) + (1-p) \end{aligned}$$

so that $(1-p) * (wp.\mathsf{S}.1) \equiv 1-p$. Since p is not 1, we can divide by $1-p$ to see that indeed $wp.\mathsf{S}.1 \equiv 1$: the recursion will terminate with probability 1 (for if p is not 1, the chance of recursing N times is p^N, which for $p < 1$ approaches 0 as N increases without bound).

We return to probabilistic termination in Section 5.

4.3 Probabilistic Correctness of Loops

As in the standard case, it is easy to show that if $[P] * I \Rrightarrow wp.\mathsf{S}.I$ then

$$I \;\Rrightarrow\; wp.(\mathbf{do}\, P \to \mathsf{S}\, \mathbf{od}).([\neg P] * I)$$

provided the loop terminates. Thus the notion of invariant carries over smoothly from the standard to the probabilistic case. This is an immediate consequence of the definition of loops as least fixed points: indeed, for the proof one simply carries out the standard reasoning almost without noticing that expectations rather than predicates are being manipulated. The precise treatment of "provided" uses weakest *liberal* pre-expectations [193, 180].

When termination is taken into account as well, we get the rule below [193].

Definition 2 (Proof rule for probabilistic loops). *For convenience, we write T for the termination probability of the loop, so that*

$$T \;\widehat{=}\; wp.(\boldsymbol{do}\, P \to \mathsf{S}\, \boldsymbol{od}).1$$

Then partial loop correctness—preservation of a loop invariant I—implies total loop correctness if that invariant I nowhere exceeds T: that is,

$$\begin{array}{lrcl} \textit{if} & [P] * I & \Rrightarrow & wp.\mathsf{S}.I \\ \textit{and} & I & \Rrightarrow & T \\ \textit{then} & I & \Rrightarrow & wp.(\boldsymbol{do}\, P \to \mathsf{S}\, \boldsymbol{od}).([\neg P] * I) \end{array}$$

Note that it is not the same to say "implies total correctness from those initial states where I does not exceed T": in fact I must not exceed T in *any* state. The weaker alternative is not sound.

We illustrate the loop rule with a simple example. Suppose we have a machine that is supposed to sum the elements of a sequence ss of N elements indexed from 0 to $N-1$, except that the mechanism for moving along the sequence occasionally moves the wrong way. A program for the machine is given in Figure 2, where the unreliable component

$$k{:=}\; k+1 \quad {}_c\!\oplus \quad k{:=}\; k-1$$

misbehaves with probability $1-c$. With what probability does the machine accurately sum the sequence, establishing

$$r \;=\; \sum ss \tag{19}$$

on termination?

```
var k: ℤ •
  r, k:= 0, 0;
  do k < N →
    r:= r + ss.k;
    k:= k + 1 c⊕ k:= k − 1        ← failure possible here
  od
```

Fig. 2. An unreliable sequence-summer

We first find the invariant. Relying on our informal discussion above, we ask the following question:

> during the loop's execution, with what probability are we in a state from which completion of the loop would establish (19)?

The answer is in the form (16)—take p to be c^{N-k}, and let I be the standard invariant

$$0 \le k \le N \quad \wedge \quad r = \sum ss[0..k)$$

Then our probabilistic invariant—call it J—is just $p * [I]$, which is to say that

> if the standard invariant holds then it is c^{N-k}, the probability of going on to successful termination; if it does not hold, then it is 0.

Having chosen a possible invariant, to check it we calculate

$$\begin{array}{ll}
& wp.\begin{pmatrix} r:= r + ss.k; \\ k:= k + 1 \ {}_c\!\oplus\ k:= k - 1 \end{pmatrix}.J \\
\equiv & wp.(r:= ss.k).(\qquad\qquad ; \text{ and } {}_c\!\oplus \\
& \quad c * wp.(k:= k+1).J \\
& \quad +(1-c) * wp.(k:= k-1).J) \\
\Leftarrow & wp.(r:= r + ss.k). \qquad \text{drop second term, and } wp.(:=) \\
& \quad c^{N-k} \ * \ \begin{bmatrix} 0 \le k+1 \le N \\ r = \sum ss[0..k) \end{bmatrix} \\
\equiv & c^{N-k} \ * \ \begin{bmatrix} 0 \le k+1 \le N \\ r + ss.k = \sum ss[0..k) \end{bmatrix} \qquad wp.(:=) \\
\Leftarrow & [k < N] * J \qquad \text{arithmetic}
\end{array}$$

where in the last step the guard $k < N$, and $k \ge 0$ from the invariant, allow the removal of $+ss.k$ from both sides of the lower equality.

A more concise rendering of the above can be given using the following convention. When reasoning "backwards", as above, the compact notation

$$\begin{array}{ll}
& PostE \\
\cdot \ \Leftarrow & PreE \qquad \text{applying } wp.\mathsf{Prog}
\end{array}$$

allows the linear "step-by-step" layout of the proof to be more easily continued. The "·" at left warns that we are asserting $PostE \Leftarrow wp.\mathsf{Prog}.PreE$ (rather than $PostE \Leftarrow PreE$ itself). Using this convention we would have written instead

$$
\begin{array}{lll}
 & J & \\
\cdot \;\equiv & \quad c * wp.(k{:=}\ k+1).J & \text{applying } wp.(k{:=}\ k+1 \ {}_c\!\oplus\, k{:=}\ k-1) \\
 & + (1{-}c) * wp.(k{:=}\ k-1).J & \\
\Leftarrow & c^{N-k} * \begin{bmatrix} 0 \le k+1 \le N \\ r = \sum ss[0..k) \end{bmatrix} & \text{drop second term; } wp.(:=) \\
\cdot \;\equiv & c^{N-k} * \begin{bmatrix} 0 \le k+1 \le N \\ r + ss.k = \sum ss[0..k) \end{bmatrix} & \text{applying } wp.(r{:=}\ r + ss.k) \\
\Leftarrow & [k < N] * J &
\end{array}
$$

Now we turn to termination: we note (informally) that the loop terminates with probability at least

$$c^{N-k} \;*\; [0 \le k \le N]$$

which is just the probability of $N-k$ correct executions of $k{:=}\ k+1$, given that k is in the proper range to start with; hence trivially $J \Rrightarrow T$ as required by the loop rule.

That concludes reasoning about the loop itself, leaving only initialisation and the post-expectation of the whole program. For the latter we see that on termination of the loop we have $[k \ge N] * J$, which indeed "implies" (is in the relation $\Rrightarrow$ to) the post-expectation $[r = \sum ss]$ as required.

Turning finally to the initialisation we finish off with

$$
\begin{array}{ll}
 & wp.(r, k{:=}\ 0, 0).J \\
\equiv & c^N \;*\; \begin{bmatrix} 0 \le 0 \le N \\ 0 = \sum ss[0..0) \end{bmatrix} \\
\equiv & c^N \;*\; [\mathbf{true}] \\
\equiv & c^N
\end{array}
$$

and our overall conclusion is therefore

$$c^N \;\Rrightarrow\; wp.(\textit{sequence-summer}).\,[r = \textstyle\sum ss]$$

just as we had hoped: the probability that the sequence is correctly summed is at least c^N.

Note the importance of the inequality $\Rrightarrow$ in our conclusion just above. It is not true that the probability of correct operation is *equal* to c^N in general, for it is certainly possible that r is correctly calculated in spite of the occasional malfunction of $k{:=}\ k+1$. The exact probability, should we try to calculate it, might depend intricately on the contents of *ss*. (It could be very involved if *ss* contained some mixture of positive and negative values.) If we were forced to calculate exact results (as in earlier work [238]), rather than just lower bounds as we did above, this method would not be at all practical.

Further examples of loops are given elsewhere [193].

5 First Case Study: Probabilistic Termination

In this case study, we treat an algorithm whose termination argument is fairly involved, showing how it is dealt with using probabilistic-variant arguments. This example has also been given an automated proof using the *pB* probabilistic extension of the *B* development method [182, 3]. For another example of "easy correctness but difficult termination", see the Probabilistic Dining Philosophers [149], [181, Section 3.2].

5.1 Introduction

Rabin's choice-coordination algorithm (explained in Sections 5.2 and 5.3 below) is an example of the use of probability for *symmetry-breaking*: identical processes with identical initial conditions must reach collectively an asymmetric state, all choosing one alternative or all choosing the other. The simplest example is a coin flipped between two people—each has equal right to win, the coin is fair, the initial conditions are thus symmetric; yet, at the end, one person has won and not the other. In this example, however, the situation is made more complex by insisting that the processes be *distributed*: they cannot share a central "coin".

Rabin's article [219] explains the algorithm he invented and relates it to a similar algorithm in nature, carried out by mites who must decide whether they should all infest the left or all the right ear of a moth, but he does not give a formal proof of its correctness. We do that here.

Section 5.3 writes the algorithm as a loop, containing probabilistic choice, and we show the loop terminates "with probability 1" in a desired state: we use invariants, to show that if it terminates it is in that state; and we use probabilistic variants to show that indeed it does terminate. "Termination with probability 1"' is the kind of termination exhibited for example by the algorithm "flip a fair coin repeatedly until you get heads, then stop". For our purposes that is as good as "normal" guarantees of termination.

In this example, the partial correctness argument is entirely standard and so does not illustrate the new probabilistic techniques. (It is somewhat involved, however, and thus interesting as an exercise in any case.) In such cases one treats probabilistic choice as nondeterministic choice and proceeds with standard reasoning, since the theory shows that any *wp*-style property proved of the "projected" nondeterministic program is valid for the original probabilistic program as well. More precisely, replacing probabilistic choice by nondeterministic choice is an anti-refinement.

The termination argument is novel however, since probabilistic variant techniques [107, 193] must be used.

5.2 Informal Description of Rabin's Algorithm

This informal description is based on Rabin's presentation [219].

A group of tourists are to decide between two meeting places: inside a (certain) church, or inside a museum. They may not communicate all at once as a group.

Each tourist carries a notepad on which he will write various numbers; outside each of the two potential meeting places is a noticeboard on which various messages will be written. Initially the number 0 appears on all the notepads and on the two noticeboards.

Each tourist decides independently (demonically) which meeting place to visit first, after which he strictly alternates his visits between them. At each visit he looks at the noticeboard there, and if it displays "here" goes inside. If it does not display "here" it will display a number instead, in which case the tourist compares that number K with the one on his notepad k and takes one of the following three actions:

if $k > K$ —The tourist writes "here" on the noticeboard (erasing K), and goes inside.

if $k = K$ —The tourist chooses K', the next even number larger than K, and then flips a coin: if it comes up heads, he increases K' by a further 1. He then writes K' on the noticeboard and on his notepad (erasing k and K), and goes to the other place. For example if K is 8 or 9, first K' becomes 10 and then possibly 11.

if $k < K$ —The tourist writes K on his notepad (erasing k), and goes to the other place.

Rabin's algorithm terminates with probability 1; and on termination all tourists will be inside, at the same meeting place.

5.3 The Program

Here we make the description more precise by giving a *pGCL* program for it (see Figure 3). Each tourist is represented by an instance of the number on his pad.

The Program Informally. Call the two places "left" and "right".

Bag *lout* (*rout*) is the bag of numbers held by tourists waiting to look at the left (right) noticeboard; bag *lin* (*rin*) is the bag of numbers held by tourists who have already decided on the left (right) alternative; number L (R) is the number on the left (right) noticeboard.

Initially there are M (N) tourists on the left (right), all holding the number 0; no tourist has yet made a decision. Both noticeboards show 0.

Execution is as follows. If some tourists are still undecided (so that *lout* (*rout*) is not yet empty), select one: the number he holds is l (r). If some tourist has (already) decided on this alternative (so that *lin* (*rin*) is not empty), this tourist does the same; otherwise there are three further possibilities:

If this tourist's number l (r) is greater than the noticeboard value L (R), then he decides on this alternative (joining *lin* (*rin*)).

If this tourist's number equals the noticeboard value, he increases the noticeboard value, copies that value and goes to the other alternative (*rout* (*lout*)).

If this tourist's number is less than the noticeboard value, he copies that value and goes to the other alternative.

$lout, rout := \llbracket 0 \rrbracket^M, \llbracket 0 \rrbracket^N;$
$lin, rin := \square, \square;$
$L, R := 0, 0;$

do $lout \neq \square \rightarrow$
  take l **from** $lout$;
  if $lin \neq \square$ **then add** l **to** lin **else**
    $l > L \;\rightarrow$ **add** l **to** lin
  $[\!]$ $l = L \;\rightarrow$ $L := L + 2 \;{}_{\frac{1}{2}}\!\oplus \overline{(L+2)}$; **add** L **to** $rout$
  $[\!]$ $l < L \;\rightarrow$ **add** L **to** $rout$
  fi

$[\!]$ $rout \neq \square \rightarrow$
  take r **from** $rout$;
  if $rin \neq \square$ **then add** r **to** rin **else**
    $r > R \;\rightarrow$ **add** r **to** rin
  $[\!]$ $r = R \;\rightarrow$ $R := R + 2 \;{}_{\frac{1}{2}}\!\oplus \overline{(R+2)}$; **add** R **to** $lout$
  $[\!]$ $r < R \;\rightarrow$ **add** R **to** $lout$
  fi
od

Fig. 3. Rabin's choice-coordination algorithm

Notation. We use the following notations in the program and in the subsequent analysis.

- $\llbracket \cdots \rrbracket$ — Bag (multiset) brackets.
- $\square$ — The empty bag.
- $\llbracket n \rrbracket^N$ — A bag containing N copies of value n.
- $b0 + b1$ — The bag formed by putting all elements of $b0$ and $b1$ together into one bag.
- **take** n **from** b — A program command: choose an element nondeterministically from non-empty bag b, assign it to n and remove it from b.
- **add** n **to** b — Add element n to bag b.
- **if** B **then** Prog **else** $\cdots$ **fi** — Execute Prog if B holds, otherwise treat $\cdots$ as a collection of guarded alternatives in the normal way.
- $\overline{n}$ — The "conjugate" value $n + 1$ if n is even, and $n - 1$ if n is odd.
- $\widetilde{n}$ — The minimum $n \; \mathit{min} \; \overline{n}$ of n and $\overline{n}$.
- $\#b$ — The number of elements in bag b.
- $x := m \;{}_p\!\oplus n$ — Assign m to x with probability p, and n to x with probability $1-p$.

Correctness Criteria. We must show that the program is guaranteed with probability 1 to terminate, and that on termination it establishes

$$\#lin = M+N \wedge rin = \square \;\vee\; lin = \square \wedge \#rin = M+N$$

That is, on termination the tourists are either all inside on the left or all inside on the right.

5.4 Partial Correctness

The arguments for partial correctness involve no probabilistic reasoning; but there are several invariants.

Three Invariants. The first invariant states that tourists are neither created nor destroyed:

$$\#lout + \#lin + \#rout + \#rin \;=\; M + N \tag{20}$$

It holds initially, and is trivially maintained.

The second invariant is

$$\begin{array}{rcl} lin, lout & \leq & R \\ rin, rout & \leq & L \end{array} \tag{21}$$

and expresses that a tourist's number never exceeds the number posted at the *other* place. By $b \leq K$ we mean that no element in the bag b exceeds the integer K. To show invariance we reason as follows:

- It holds initially.
- Since L, R never decrease, it can be falsified only by adding elements to the bags.
- Adding elements to lin, rin cannot falsify it, since those elements come from $lout, rout$.
- The only commands adding elements to $lout, rout$ are

  $$\textbf{add}\, L\, \textbf{to}\, rout \quad \text{and} \quad \textbf{add}\, R\, \textbf{to}\, lout$$

  and they maintain it trivially.

Our final invariant for partial correctness is

$$\begin{array}{ll} \mathsf{max}\, lin > L & \text{if } lin \neq \Box \\ \mathsf{max}\, rin > R & \text{if } rin \neq \Box \end{array} \tag{22}$$

expressing that if any tourist has gone inside, then at least one of the tourists inside must have a number exceeding the number posted outside.

By symmetry we need only consider the left (lin) case. The invariant holds on initialisation (when $lin = \Box$); and inspection of the program shows that it is trivially established when the first value is added to lin since the command concerned

$$l > L \;\rightarrow\; \textbf{add}\, l\, \textbf{to}\, lin$$

is executed when $lin = \Box$ to establish $lin = [\![\, l \,]\!]$ for some $l > L$.

Since elements never leave lin, it remains non-empty and $\mathsf{max}\, lin$ can only increase; finally L cannot change when lin is non-empty.

On Termination... With these invariants we can show that on termination (if it occurs) we have $lout = rout = \square$—in fact with invariant (20) we need only

$$lin = \square \ \lor \ rin = \square$$

Assuming for a contradiction that both *lin* and *rin* are non-empty, we then have from invariants (21) and (22) the inequalities

$$L \geq \mathsf{max}\, rin \ > \ R \ \geq \ \mathsf{max}\, lin \ > \ L$$

which give us the required impossibility.

5.5 Showing Termination: The Variant

For termination we need probabilistic arguments, since it is easy to see that no standard variant will do: suppose that the first $M + N$ iterations of the loop take us to the state below, differing from the initial state only in the use of 4's rather than 0's.

$$\begin{aligned} lout, rout &= \llbracket 4 \rrbracket^M, \llbracket 4 \rrbracket^N \\ lin, rin &= \square, \square \\ L, R &= 4, 4 \end{aligned}$$

All coin flips came up heads, and each tourist had exactly two turns. Since the program contains no absolute comparisons, we are effectively back where we started: the program checks only whether various numbers are greater than others, not what the numbers actually are. Because of that, there can be no standard variant that decreased on every step we took.

So is not possible to prove termination using a standard variant whose strict decrease is guaranteed. Instead we appeal to the following rule [107, 193, 181]:

Definition 3 (Probabilistic variant rule). *If an integer-valued function of the program state—a* probabilistic variant*—can be found that*

- *is bounded above,*
- *is bounded below and*
- *with probability at least p is decreased by the loop body, for some fixed non-zero p,*

then with probability 1 the loop will terminate. (Note that the invariant and guard of the loop may be used in establishing the three properties.)

The rule differs from the standard one in two respects: the variant must be bounded above (as well as below); and it is not guaranteed to decrease, but rather does so only with some probability bounded away from 0. Note that the probability of decrease may differ from state to state, but the point of "bounded away from zero"—distinguished from simply "not equal to zero"—is that over

an infinite state space the various probabilities cannot be arbitrarily small. Over a finite state space there is no distinction.

To find our variant, we note that the algorithm exhibits two kinds of behaviour: the shuttling back-and-forth of the tourists, between the two meeting places (small scale); and the pattern of the two noticeboard numbers L, R as they increase (large scale). Our variant therefore will be "lexicographic", one within another: the small-scale *inner* variant will deal with the shuttling, and the large-scale *outer* variant will deal with L and R.

Inner Variant: Tourists' Movements. The aim of the inner variant is to show that the tourists cannot shuttle forever between the sites without eventually changing one of the noticeboards. Intuition suggests that indeed they cannot, since every such movement increases the number on some tourist's notepad, and from invariant (21) those numbers are bounded above by $L \mathbin{\mathsf{max}} R$.

The inner variant is based on that idea. For neatness we make it increasing rather than decreasing, which is of no consequence since we have taken care to ensure that it is bounded above and below by fixed values, independent of L and R—we could always subtract it from the upper bound to convert it back to decreasing. The independence from L, R is important, given our variant rule, because L and R can themselves increase without bound. We define $V0$ to be

$$\begin{array}{ll} & \#\llbracket x\colon lout{+}rout \mid x \geq L \rrbracket \\ + & \#\llbracket x\colon lout{+}rout \mid x \geq R \rrbracket \\ + & 3 \times \#(lin{+}rin) \end{array} \tag{23}$$

This is bounded above by $3(M{+}N)$, because

$$(23) \leq 2\#(lout{+}rout) + 3\#(lin{+}rin) \leq 3\#(lout{+}rout{+}lin{+}rin) = 3(M{+}N)$$

where the last equality is supplied by the invariant (20). Since the outer variant will deal with changes to L and R, in checking the increase of $V0$ we can restrict our attention to those parts of the loop body that leave L, R fixed—and we show in that case that the variant must increase on every step:

- If $lin \neq \Box$ then an element is removed from $lout$ ($V0$ decreases by at most 2) and added to lin (but then $V0$ increases by 3); the same reasoning applies when $l > L$.
- If $l = L$ then L will change; so we need not consider that. (It will be dealt with by the outer variant.)
- If $l < L$ then $V0$ increases by at least 1, since l is replaced by L in $lout{+}rout$—and (before) $l \not\geq L$ but (after) $L \geq L$.

The reasoning for $rout$, on the right, is symmetric.

Outer Variant: Changes to L and R. For the outer variant we need further invariants; the first is

$$\widetilde{L} - \widetilde{R} \;\in\; \{-2, 0, 2\} \tag{24}$$

stating that the notice-board values can never be "too far apart". It holds initially; and, from invariant (21), the command

$$L{:=}\ L + 2 \ {}_{\frac{1}{2}}\!\oplus \overline{(L + 2)}$$

is executed only when $L \leq R$, thus only when $\widetilde{L} \leq \widetilde{R}$, and has the effect

$$\widetilde{L}{:=}\ \widetilde{L} + 2$$

Thus we can classify L, R into three sets of states:

- $\widetilde{L} = \widetilde{R} - 2 \ \vee\ \widetilde{L} = \widetilde{R} + 2$—write $L \not\cong R$ for those states.
- $\overline{L} = R$ (equivalently $L = \overline{R}$)—write $L \mathrel{\widetilde{=}} R$.
- $L = R$.

Then we note that the underlying iteration of the loop induces state transitions as follows. (We write $\langle L = R\rangle$ for the set of states satisfying $L = R$, and so on; nondeterministic choice is indicated by $\sqcap$; the transitions are indicated by $\rightarrow$.)

$$\begin{array}{lcl}
\langle L \not\cong R\rangle & \rightarrow & \langle L \not\cong R\rangle \ \sqcap\ \langle L = R\rangle \ {}_{\frac{1}{2}}\!\oplus \langle L \mathrel{\widetilde{=}} R\rangle \\
\langle L = R\rangle & \rightarrow & \langle L = R\rangle \ \sqcap\ \langle L \not\cong R\rangle \\
\langle L \mathrel{\widetilde{=}} R\rangle & \rightarrow & \langle L \mathrel{\widetilde{=}} R\rangle
\end{array}$$

To explain the absence of a transition leaving states $\langle L \mathrel{\widetilde{=}} R\rangle$ we need yet another invariant

$$\overline{L} \notin \mathit{rout} \ \wedge\ \overline{R} \notin \mathit{lout} \tag{25}$$

It holds initially, and cannot be falsified by the command **add** L **to** *rout*, because $\overline{L} \neq L$. That leaves the command $L{:=}\ L + 2 \ {}_{\frac{1}{2}}\!\oplus \overline{(L + 2)}$; but in that case, from (21), we have

$$\mathit{rout} \ \leq\ L \ <\ L + 2, \overline{(L + 2)} \ =\ \overline{\overline{(L + 2)}}, \overline{(L + 2)}$$

so that in neither case does the command set L to the conjugate of a value already in *rout*.

Thus with (25) we see that execution of the only alternatives that change L, R cannot occur if $L \mathrel{\widetilde{=}} R$, since, for example, selection of the guard $l = L$ implies $L \in \mathit{lout}$, impossible if $L \mathrel{\widetilde{=}} R$ and $\overline{R} \notin \mathit{lout}$.

For the outer variant we therefore define *V1* to be

$$\begin{array}{ll}
2, & \text{if } L = R \\
1, & \text{if } L \not\cong R \\
0, & \text{if } L \mathrel{\widetilde{=}} R
\end{array} \tag{26}$$

and note that whenever L or R changes, the quantity *V1* decreases with probability at least *1/2*.

The Two Variants Together. If we put the two variants together lexicographically, with the outer variant *V1* being the more significant, then the composite satisfies all the conditions required by the probabilistic variant rule. In particular it has probability at least *1/2* of strict decrease on *every* iteration of the loop. Remember that the inner variant increases rather than decreases—we subtract it from $3(M+N)$ to make it decrease.

Thus the algorithm terminates with probability 1—and we are done.

Exercise 3. Argue informally that the loop

$$
\begin{array}{l}
c := H \;{}_p\!\oplus\; T;\\
\textbf{do}\; c \neq H \;\rightarrow\\
\quad c := H \;{}_p\!\oplus\; T;\\
\textbf{od}
\end{array}
$$

terminates with probability one provided $p > 0$. Then prove it formally by finding a variant function and using the PROBABILISTIC VARIANT RULE.

Exercise 4. Show that the loop

$$
\begin{array}{l}
c, d := H, H;\\
\textbf{do}\; c = d \;\rightarrow\\
\quad c := H \;{}_p\!\oplus\; T;\\
\quad d := H \;{}_p\!\oplus\; T\\
\textbf{od}
\end{array}
$$

establishes $c = H$ on termination with probability *1/2* for *any* p, provided $0 < p < 1$. (Note that the two coins have the same bias, although it is almost arbitrary: think of it as the same coin flipped repeatedly, where in the loop guard we are comparing the last two results.) *Hint:* Consider the invariant (a real-valued function) defined by the matrix

$$\begin{pmatrix} \frac{1}{2} & 1 \\ 0 & \frac{1}{2} \end{pmatrix}$$

where c selects the row and d selects the column. Do not forget the variant.

Exercise 5. Let $\overline{H}$ be T and $\overline{T}$ be H, so that $d := \overline{d}$ simply turns d over. Show that the loop

$$
\begin{array}{l}
c, d := H, H;\\
\textbf{do}\; c = H \;\rightarrow\\
\quad c := H \;{}_{1/2}\!\oplus\; T;\\
\quad d := \overline{d}\\
\textbf{od}
\end{array}
$$

establishes $d = H$ on termination with probability exactly *1/3*. (This is a good way of dealing with the "one ice-cream, three sons" problem.) *Hint:* Consider the invariant

$$\begin{pmatrix} \frac{1}{3} & \frac{2}{3} \\ 1 & 0 \end{pmatrix}$$

6 Second Case Study: Approximated Probabilities, Abstraction and Refinement

In this case study, we give a small example of a probabilistic program developed in two stages, linked by abstraction and refinement, and in which the issue of "approximate" probabilities is highlighted. This section is based on an example in Hurd's thesis, where, however, the probabilities are exact [125]; we treat the exact case elsewhere [196].

For practical purposes we suppose a source of randomness is available as a stream of unbiased random bits; however many applications' correctness relies on more elaborate distributions. Those distributions can be generated by using various sampling methods; here (Figure 4) we consider a small program which uses (nearly) unbiased bits to generate a (nearly) uniform choice over a positive number N of alternatives. That is, we imagine we have access to a stream of bits, each equally likely to be 0 or 1, but we need to choose uniformly between N alternatives (rather than just 2, which the bits could do directly). We want to write a program to carry this out.

$$
\begin{array}{l}
\{\,[0 \le K < N]/N_\varepsilon\,\} \\
\textbf{var}\ k : \mathbb{N} \bullet \\
\quad k := N; \\
\quad \textbf{do}\ k \ge N\ \rightarrow \\
\qquad \textbf{var}\ n : \mathbb{N}; \\
\qquad k, n := 0, N-1; \\
\qquad \textbf{do}\ n \neq 0\ \rightarrow \\
\qquad\quad k := 2k\ {}_{\frac{1}{2}-\varepsilon}\!\oplus_{\frac{1}{2}-\varepsilon}\ k := 2k+1; \\
\qquad\quad n := n \operatorname{div} 2 \\
\qquad \textbf{od} \\
\quad \textbf{od} \\
\{\,[k=K]\,\}
\end{array}
$$

The inner loop selects k almost uniformly such that $0 \le k < \$N$, where $\$N$ is the least power-of-two no less than N. The outer loop accepts that choice only if $k < N$; otherwise the inner loop is repeated. The effect overall is to select k almost uniformly so that $0 \le k < N$.

The pre- and post-expectation annotations express that for any K the probability of achieving $k = K$ on termination is at least $1/N_\varepsilon$ if $0 \le K < N$ (and at least zero otherwise), where $N_\varepsilon \ge N$. The "excess" $N_\varepsilon - N$ quantifies the inaccuracy, and should tend to zero as ε does.

Fig. 4. Almost-uniform selection algorithm

6.1 Approximation Via Nondeterminism

Suppose we have access to a stream of bits b each of which is independently unbiased but only to within some tolerance ε, by which we mean that the probability of a 1 (or 0) is only within ε of $1/2$ on each occasion. In *pGCL* we would express this by using the statement

$$b{:=}\ 0\ {}_{\frac{1}{2}-\varepsilon}\!\oplus_{\frac{1}{2}-\varepsilon}\ b{:=}\ 1$$

That is, we are using this statement to model what probably is a piece of hardware, and the ε in the probabilistic-choice operator represents how accurate we have observed this hardware to be: if it were completely accurate ($\varepsilon = 0$) then the statement would be just a "coin flip" of b.

In fact because we will always be "shifting left" our random bits into a bit-string represented by k, we will use the statement

$$k{:=}\ 2k\ {}_{\frac{1}{2}-\varepsilon}\!\oplus_{\frac{1}{2}-\varepsilon}\ k{:=}\ 2k+1 \tag{27}$$

at the point where we access the random bit-stream. We recall that, for $p+q \leq 1$ in general, by $\mathsf{This}\ {}_p\!\oplus_q\ \mathsf{That}$ we mean the nondeterministic combination

$$\mathsf{This}\ {}_p\!\oplus\ \mathsf{That}\ \sqcap\ \mathsf{That}\ {}_q\!\oplus\ \mathsf{This} \tag{28}$$

of the two programs that (on the left) executes This with probability p (and That with probability $1-p$), and (on the right) executes That with probability q (and This with probability $1-q$).

The operational semantics of *pGCL* identifies Program (28) with one that chooses This (rather than That) with any probability r satisfying $p \leq r \leq 1-q$, because the space of possible program behaviours is "convex-closed" [131, 197, 181], reflecting that nondeterministic choices can be resolved to arbitrary probabilistic ones. Thus the program fragment (27) "flips the coin" with any probability r satisfying $1/2-\varepsilon \leq r \leq 1/2+\varepsilon$, which captures our intended meaning above of "unbiased only to within some tolerance ε". The value of r can vary between separate executions of the fragment, and we recall that it is adversarial in the sense that its choice is treated as worst-case by our program logic, in effect determined by a "demon" whose aim is to make our program as unlikely as possible to produce a uniform distribution.

What is the aim of the program? It is to set k to some value K in the range $[0, N)$, and the effect of the introduced nondeterminism will be to make it less likely to do that than the $1/N$ we would expect of an exactly uniform distribution.

6.2 Overall Analysis Strategy

We will give a conservative analysis of the program, which is safe but slightly pessimistic, on the grounds that it is simpler than an exact analysis would be, and that for small biases the assurance it gives us is good enough. Informally our reasoning will be as follows.

The inner loop chooses a number k in the range $[0, \$N)$, where $\$N$ is the smallest power-of-two no less than N; it does that by assembling a $\flat N$-bit number via a series of calls to the random bit generator, where $\flat N$ is the minimum number of bits sufficient to represent any number in the given range.

Because the bit generator is biased, however, the minimum guaranteed probability of producing any particular K with $0 \leq K < \$N$ is only $1/\delta^{\flat N}$ (instead of $1/2^{\flat N}$, that is $1/\$N$), where for convenience we set $1/\delta := 1/2 - \varepsilon$. Thus—informally—the maximum guaranteed probability x of producing K in the given range satisfies

$$x \;\geq\; 1/\delta^{\flat N} + (\$N - N)x/\delta^{\flat N} \tag{29}$$

where the second term is a lower bound on the probability that the inner loop chooses a k that is "too big", that is with $k \geq N$, thus forcing a subsequent iteration. It is only a lower bound because the actual probability of achieving $N \leq k < \$N$ is usually higher, given the way in which k is constructed.

For example, note that although the minimum guaranteed probability of setting k to $\$N{-}1$ is $1/\delta^{\flat N}$, and similarly to $\$N{-}2$, the probability of achieving either, that is $\$N{-}2 \leq k < \$N$, is in fact $1/\delta^{\flat N-1}$ because in that case only the first $\flat N{-}1$ bits of k are constrained. That is more than the sum $1/\delta^{\flat N} + 1/\delta^{\flat N}$ given by considering the two values separately (unless there is no bias, that is unless $\delta = 2$ exactly).

This is the essence of our abstraction, that we ignore the bit-by-bit structure of k in order to get a good-enough result by simpler means. Solving (29) gives

$$x \;\geq\; 1/(N + (\delta^{\flat N} - \$N))$$

which identifies the quantity $\delta^{\flat N} - \$N$ as a sort of "excess" E which lowers the probability from the uniform $1/N$ to some $1/N_\varepsilon$ where $N_\varepsilon := N + E$.

6.3 Proofs for Inner Loop

We analyse the program in two levels, first the inner loop and then the outer loop.

As well as the definitions above, we let $\overline{\flat}n$ be the number of bits used in the binary representation of n, and let $\overline{\$}n$ be the smallest power-of-two strictly exceeding n, so that $2^{\overline{\flat}n} = \overline{\$}n$ (which makes it clear that $\overline{\flat}0 = 0$). These "strict" definitions have slightly better algebraic properties than the "non-strict" ones above, and simplify the calculation. In fact $\overline{\$}N = \$(N{+}1)$ of course, so we are just avoiding a mess of brackets and $+1$'s.

For the inner loop, where the selection range is $\$N$, a nice power of two, it's a reasonable guess that the effect of the bias introduced by the nondeterminism will be to reduce any particular K's chances from $1/\$N$, that is $1/2^{\flat N}$, down to $1/\delta^{\flat N}$—and we note (reassuringly) that when $\varepsilon = 0$ those two probabilities are equal. That suggests the overall precondition for our approximating inner loop, and similar considerations suggest an invariant for it: the resulting annotated loop is shown in Figure 5. In the following, we justify the annotations.

On Initialisation. This is straightforward; we reason

$$
\begin{array}{ll}
& \left[k(\overline{\$}n) \le K < (k{+}1)(\overline{\$}n)\right] / \delta^{\overline{\flat}n} & \text{invariant} \\
\cdot \equiv & & \text{applying } wp.(k, n{:=}\ 0, N{-}1) \\
& \left[0(\overline{\$}(N{-}1)) \le K < (0{+}1)(\overline{\$}(N{-}1))\right] / \delta^{\overline{\flat}(N-1)} & \\
\equiv & [0 \le K < \$N] / \delta^{\flat N} & \text{arithmetic gives pre-expectation}
\end{array}
$$

While Iterating. We reason backwards from the end of the loop body towards its beginning. The novelty here is the demonic nondeterminism in $\frac{1}{\delta} \oplus_{\frac{1}{\delta}}$ which we interpret as at (28), leading to the use of **min** as indicated by the semantics given at in (7).

$$
\begin{array}{ll}
& \left[k(\overline{\$}n) \le K < (k{+}1)(\overline{\$}n)\right] / \delta^{\overline{\flat}n} & \text{invariant} \\
\cdot \equiv & & \text{applying } wp.(n{:=}\ n\mathsf{div}2) \\
& \left[k(\overline{\$}(n\mathsf{div}2)) \le K < (k{+}1)(\overline{\$}(n\mathsf{div}2))\right] / \delta^{\overline{\flat}(n\mathsf{div}2)} & \\
\cdot \equiv & & \text{applying } wp.(k{:=}\ 2k \ {}_{\frac{1}{\delta}}\!\oplus_{\frac{1}{\delta}}\ 2k+1) \\
& 1/\delta * \left[\begin{array}{l}(2k)(\overline{\$}(n\mathsf{div}2)) \\ \le K \\ < ((2k){+}1)(\overline{\$}(n\mathsf{div}2))\end{array}\right] / \delta^{\overline{\flat}(n\mathsf{div}2)} & \\
& + \ (1{-}1/\delta) * \left[\begin{array}{l}(2k{+}1)(\overline{\$}(n\mathsf{div}2)) \\ \le K \\ < ((2k{+}1){+}1)(\overline{\$}(n\mathsf{div}2))\end{array}\right] / \delta^{\overline{\flat}(n\mathsf{div}2)} & \\
& \textbf{min} & \\
& (1{-}1/\delta) * \left[\begin{array}{l}(2k)(\overline{\$}(n\mathsf{div}2)) \\ \le K \\ < ((2k){+}1)(\overline{\$}(n\mathsf{div}2))\end{array}\right] / \delta^{\overline{\flat}(n\mathsf{div}2)} & \\
& + \quad 1/\delta * \left[\begin{array}{l}(2k{+}1)(\overline{\$}(n\mathsf{div}2)) \\ \le K \\ < ((2k{+}1){+}1)(\overline{\$}(n\mathsf{div}2))\end{array}\right] / \delta^{\overline{\flat}(n\mathsf{div}2)} & \\
\Leftarrow & & \text{arithmetic; } 1/\delta \le 1{-}1/\delta \\
& \left[\begin{array}{l}(2k)(\overline{\$}(n\mathsf{div}2)) \\ \le K \\ < (2k{+}1)(\overline{\$}(n\mathsf{div}2))\end{array}\right] / \delta^{\overline{\flat}(n\mathsf{div}2)+1} & \\
& + \left[\begin{array}{l}(2k{+}1)(\overline{\$}(n\mathsf{div}2)) \\ \le K \\ < (2(k{+}1))(\overline{\$}(n\mathsf{div}2))\end{array}\right] / \delta^{\overline{\flat}(n\mathsf{div}2)+1} & \\
\Leftarrow & [n \ne 0] * & 2\overline{\$}(n\mathsf{div}2) = \overline{\$}n;\ \overline{\flat}(n\mathsf{div}2){+}1 = \overline{\flat}n \\
& \left(\left[k(\overline{\$}n) \le K < (2k{+}1)(\overline{\$}(n\mathsf{div}2))\right] / \delta^{\overline{\flat}n}\right. & \\
& \left. + \left[(2k{+}1)(\overline{\$}(n\mathsf{div}2)) \le K < (k{+}1)(\overline{\$}n)\right] / \delta^{\overline{\flat}n}\right) &
\end{array}
$$

$\{\,[0 \le K < \$N] / \delta^{\flat N}\,\}$
$k, n := 0, N-1;$
$\{\,\left[k(\overline{\$}n) \le K < (k+1)(\overline{\$}n)\right] / \delta^{\overline{\flat} n}\,\}$
do $n \neq 0 \;\rightarrow$
$\{\,\left[k(\overline{\$}n) \le K < (k+1)(\overline{\$}n)\right] / \delta^{\overline{\flat} n}\,\}$
$k := 2k \;{}_{\frac{1}{\delta}}\!\oplus\!{}_{\frac{1}{\delta}}\; 2k+1;$
$\{\,\left[k\overline{\$}(n \operatorname{div} 2) \le K < (k+1)\overline{\$}(n \operatorname{div} 2)\right] / \delta^{\overline{\flat}(n \operatorname{div} 2)}\,\}$
$n := n \operatorname{div} 2$
$\{\,\left[k(\overline{\$}n) \le K < (k+1)(\overline{\$}n)\right] / \delta^{\overline{\flat} n}\,\}$
od
$\{\,[k=K]\,\}$

Fig. 5. Approximating inner loop with annotations

$$\Leftarrow \qquad \text{merging inequalities gives guard and invariant}$$
$$[n \neq 0] * \left[k(\overline{\$}n) \le K < (k+1)(\overline{\$}n)\right] / \delta^{\overline{\flat} n}$$

On Termination. This is immediate; we have

$$\begin{aligned}
&[n = 0] * \left[k(\overline{\$}n) \le K < (k+1)(\overline{\$}n)\right] / \delta^{\overline{\flat} n} && \text{negated guard and invariant}\\
&\Rrightarrow \left[k(\overline{\$}0) \le K < (k+1)(\overline{\$}0)\right] / \delta^{\overline{\flat} 0} && \text{arithmetic}\\
&\Rrightarrow [k \le K < (k+1)] && \overline{\$}0 = 1\\
&\Rrightarrow [k = K] && k, K \in \mathbb{N} \text{ gives post-expectation}
\end{aligned}$$

6.4 The Algebra of Abstractions

We now use what we have proved about the inner loop to deduce a property for use in the outer loop. As mentioned in Section 6.2, we are taking a conservative view (though sound) to simplify the calculations. We write $PostE_k^K$ for $PostE$ with variable k replaced by constant K, and use *sub-linearity* from (11) to reason

$$\left.\begin{aligned}
&wp.\mathsf{Inner}.PostE\\
&\Leftarrow && \text{arithmetic, } wp.\mathsf{Inner} \text{ monotonic}\\
&\quad wp.\mathsf{Inner}.\left(\textstyle\sum_{0 \le K < \$N}\; PostE_k^K * [k = K]\right)\\
&\Leftarrow && \textit{sublinearity} \text{ (11) used } \$N-1 \textit{times}\\
&\quad \left(\textstyle\sum_{0 \le K < \$N}\; PostE_k^K * wp.\mathsf{Inner}.[k = K]\right)\\
&\Leftarrow \left(\textstyle\sum_{0 \le K < \$N}\; PostE_k^K * [0 \le K < \$N] / \delta^{\flat N}\right) && \text{Section 6.3}\\
&\Leftarrow \left(\textstyle\sum_{0 \le K < \$N}\; PostE_k^K\right) / \delta^{\flat N} && \text{within summation } [0 \le K < \$N] = 1\\
&\Leftarrow \left(\textstyle\sum_{0 \le k < \$N}\; PostE\right) / \delta^{\flat N} && \text{change bound variable } K \text{ to } k
\end{aligned}\right\} \quad (30)$$

If we now took (30) as the *wp-definition* of Inner, rather than merely a property of it, we would effectively have an abstraction (*i.e.* an anti-refinement) of the actual inner loop. As usual for refinement, any conclusion we draw about Inner (such as its contribution to the correctness of the outer loop, argued below) are valid for the actual inner loop as well.

6.5 Proofs for Outer Loop

The complete annotations for our outer loop are given in Figure 6. Since it has nonzero probability $N/\delta^{\flat N}$ of termination on each iteration, its overall termination probability is one. We leave N_ε undetermined for now: at the appropriate moment in the proof we will discover what it must be.

$$
\begin{array}{l}
\{\,[0{\le}K{<}N]/N_\varepsilon\,\}\\
k{:=}\ N;\\
\{\,[k{=}K]\lhd k{<}N\rhd[0{\le}K{<}N]/N_\varepsilon\,\}\\
\textbf{do}\ \ k\ge N\ \ \to\\
\quad\{\,[0{\le}K{<}N]/N_\varepsilon\,\}\\
\quad \mathsf{Inner}\\
\quad\{\,[k{=}K]\lhd k{<}N\rhd[0{\le}K{<}N]/N_\varepsilon\,\}\\
\textbf{od}\\
\{\,[k{=}K]\,\}
\end{array}
$$

Fig. 6. Outer loop, with inner loop abstracted

The on-initialisation and on-termination arguments are trivial. The while-iterating argument is as follows:

$$
\begin{array}{lll}
& [k=K]\lhd k<N\rhd[0\le K<N]/N_\varepsilon & \text{invariant}\\
\cdot\ \equiv & (\ \sum_{0\le k<N}\ [k=K] & \text{applying } wp.\mathsf{Inner} \text{ from (30), and } \$N\ge N\\
& \quad +\sum_{N\le k<\$N}\ [0\le K<N]/N_\varepsilon & \\
&)/\delta^{\flat N} & \\
\equiv & (\ [0\le K<N] & \text{arithmetic}\\
& \quad +(\$N-N)\,[0\le K<N]/N_\varepsilon & \\
&)/\delta^{\flat N} & \\
\Leftarrow & [0\le K<N]/N_\varepsilon & \text{see below}\\
\Leftarrow & [k=K]\lhd k<N\rhd[0\le K<N]/N_\varepsilon & \text{assuming guard } k\ge N
\end{array}
$$

The deferred justification in the second-last step is the information we need to determine N_ε: it is sufficient to have

$$(1+(\$N-N)/N_\varepsilon)/\delta^{\flat N}\ \ \ge\ \ 1/N_\varepsilon$$

that is $N_\varepsilon \ge N+(\delta^{\flat N}-\$N) = N+E$, say.

6.6 Discussion

The "excess" E can be regarded as a price we must pay for the bias in our random-bit source: because $\delta \geq 2$ and so $\delta^{\flat N} \geq \$N$, it is never negative; and, as expected, if the bias ε is zero then δ is 2 exactly, making the excess E zero as well.

Another special case is when N is an exact power of two, whence $N = \$N$ and so $N_\varepsilon \geq \delta^{\flat N}$, again as one would expect.

As an example of the general case, we suppose our bit-source is up to 1% biased either way, and we are using it to make uniform selections from 10 alternatives; then we would have

$$\begin{array}{rrcll} & \varepsilon & = & .01 & \\ \text{and} & N & = & 10\,, & \\ & & & & \\ \text{hence} & \delta & = & 1/0.49 & \approx \quad 2.04\,, \\ & E & \approx & 2.04^4 - 16 & \approx \quad 1.35 \\ \text{and} & N_\varepsilon & \geq & \sim 11.35 & \end{array}$$

so that our conservative estimate gives each of our ten choices a guaranteed probability of just under one-in-eleven of being chosen. A more exact but informal analysis in our earlier style would look at the actual bit patterns as follows. The probability of setting $k := K$ within the inner loop is at least $1/\delta^4$; otherwise there is a guaranteed probability that k will be set "high" so that the inner loop will be tried again, as in this table:

$$\text{inner-loop outcomes where } k \text{ is "high"} \quad \left\{ \begin{array}{ll} \left.\begin{array}{ll} 10- & 1010 \\ 11- & 1011 \end{array}\right\} & \text{probability } 1/\delta^3 \\ \left.\begin{array}{ll} 12- & 1011 \\ 13- & 1100 \\ 14- & 1101 \\ 15- & 1111 \end{array}\right\} & \text{probability } 1/\delta^2 \end{array} \right.$$

This leads to the inequality

$$x \;\geq\; 1/\delta^4 + (1/\delta^2 + 1/\delta^3)x$$

giving $N_\varepsilon \geq \sim 11.14$—which is not much improvement for the extra trouble. In general, exact calculations for the high-outcome probabilities would be unpleasant.

7 Conclusion

It seems that a little generalisation can go a long way: Kozen's use of expectations and the definition of ${}_p\oplus$ as a weighted average [140] is all that is needed for a simple probabilistic semantics, albeit one lacking abstraction. Then He's *sets* of distributions [131] and our *min* for demonic choice together with the fundamental

property of sublinearity [197] take us the rest of the way, allowing abstraction and refinement to resume their central role—this time in a probabilistic context. And as Sections 4 and 5 illustrate, many of the standard reasoning principles carry over almost unchanged.

Being able to reason formally about probabilistic programs does not of course remove *per se* the complexity of the mathematics on which they rely: we do not now expect to find astonishingly simple correctness proofs for all the large collection of randomized algorithms that have been developed over the decades [201]. Our contribution—at this stage—is to make it possible in principle to locate and determine reliably what are the probabilistic/mathematical facts the construction of a randomized algorithm needs to exploit... which is of course just what standard predicate transformers do for conventional algorithms.

In practice however, one is interested not only in certain and correct termination of random algorithms, but in how long they take to do so. Such algorithms' performance cannot be put within bounds in the normal way: instead, one speaks of the *expected* time to termination, how long "on average" should one expect the algorithm to take. When the algorithm is also nondeterministic (as in Rabin's, where no assumptions are made about the order or frequency of the tourists' travels), the estimate would have to be "worst-case" expected.

And there is the larger issue of probabilistic modules, and the associated concern of probabilistic data refinement. That is a challenging problem, with lots of surprises: using our new tools we have already seen that probabilistic modules sometimes do not mean what they seem [183], and that equivalence or refinement between them depends subtly on the power of demonic choice and its interaction with probability.

Other areas in which probabilistic semantics is relevant include concurrent- and relational models. For the former there is an extremely large literature on probabilistic labelled transition systems in the *CCS* style [148, 248, for example], with (as usual) an emphasis on bisimulation; the denotational approach favoured by *CSP* is represented by a smaller but no less elegant body of research [234, 172, 198, 190]. A connection between the latter and our sequential approach can be made via action systems [195].

Probabilistic semantics has attractions for relational programming as well, where programs are represented directly as relations between initial and final states (or as predicates over them) as in the UTP (see Chapter 6). An attractive generalisation is to replace the Booleans by real values, so that the "extended relation" produces directly the probability of making a transition from a given initial to a given final state; a challenge is to do this without losing the ability to describe demonic nondeterminism as well.

Exercise 6. Let n be a natural number. The loop

$c, n := H, 0;$
do $c = H \rightarrow$
  $c := H \;{}_{1/2}\!\oplus\; T;$
  $n := n{+}1$
od

terminates with probability one, and can produce any positive integer as the final value of n. Thus it is not *image-finite*, a condition normally considered to be a "well-behavedness" criterion for sequential programs, and guaranteeing their continuity.

But what do we mean by continuity in this context? Is the above program continuous after all? If it is, can you give an example of a *pGCL* program that is not?

Exercise 7. A more immediate approach to probabilistic semantics might be to retain Boolean logic while extending the *wp* modality to include an explicit lower-bound probability: thus

$$wp_p.\mathsf{S}.P \tag{31}$$

would describe those initial states from which termination of S in a final state satisfying P was guaranteed with probability at least p. (Free variables in p, if present, would be resolved in the initial state.) Thus we could write for example $wp_{\frac{1}{2}}.(c{:=}\ H \,{}_{1/2}\!\oplus T).(c = H)$ to describe those states from which $c{:=}\ H \,{}_{1/2}\!\oplus T$ is guaranteed to establish $c = H$ with probability at least *1/2* (which is in fact all states).

1. Write the precondition $wp_p.\mathsf{S}.P$ in our logic of expectations, thus showing that the latter is at least as expressive.
2. By considering the two programs

$$\begin{array}{ll} & x{:=}\ A \sqcap (x{:=}\ B \,{}_{1/2}\!\oplus x{:=}\ C) \\ \text{and} & (x{:=}\ A \sqcap x{:=}\ C) \,{}_{1/2}\!\oplus (x{:=}\ B \sqcap x{:=}\ C)\,, \end{array}$$

show that in fact (31) is not expressive enough.

Real-Time and Fault-Tolerant Systems

Zhiming Liu[1] and Mathai Joseph[2]

[1] International Institute for Software Technology
United Nations University
Macao SAR, China
[2] Tata Research Development and Design Centre
Pune, India

In this chapter, we show that functional and many non-functional properties of a real-time system, such as schedulability, or proving that its implementation meets its timing constraints, can be verified in a similar way. Likewise, the fault-tolerance of a system can be proved using the same techniques. We use a single notation and model and take a unified view of the functional and non-functional properties of programs. A simple transformational method is used to combine these properties [167, 168]. We show how the theory of concurrency, fault-tolerance, real-time and scheduling can be built on the theories of sequential programming, such as those of Dijkstra's calculus of weakest preconditions [81], Hoare Logic [114], Morgan's refinement calculus [192] and Hoare and He's UTP [117]. These theories are discussed and used in Chapter 4 and Chapter 6.

Section 1 gives an informal account of real-time systems. Section 2.1 presents a historic background on formal techniques in real-time and fault-tolerance. Section 2.2 gives an outline of the approach used in this chapter. Section 3 introduces the computational model and the Temporal Logic of Actions [144] used for program specification, verification and refinement. In Section 4, we show how physical faults are specified, how fault-tolerance is achieved by transforming a non-fault-tolerant program, and how fault-tolerance is verified and refined. Section 5 extends the method given in Section 3 for the specification and verification of real-time programs. In Section 6 we combine the techniques used for fault-tolerance and real-time. Section 7 shows how real-time scheduling policies can be specified and combined with the program specification for verification of schedulability of a program. Proof rules for feasibility and fault-tolerant feasibility are also developed and it is shown how methods and results from scheduling theory can be formally verified and used. The notation, model and techniques are illustrated using a simple processor-memory interface program.

1 Real-Time Systems: An Informal Account

A real-time system must meet functional and timing properties when implemented on a chosen hardware platform. Some timing properties can be derived from the specification of the system and others from the design choices made in the implementation. Yet other properties can be determined only by examining the timing characteristics of the implementation.

A. Cavalcanti, A. Sampaio, and J. Woodcock (Eds.): PSSE 2004, LNCS 3167, pp. 156–219, 2006.

Real-time systems often need to meet critical safety requirements under a variety of operating conditions. One factor that then needs attention is the ability of the system to overcome the effects of faults that may occur in the system. Such faults are usually defined in terms of a fault-model. The degree of *fault-tolerance* of the system must be established in terms of the fault-model and the effect of faults upon the execution of the system.

Consider a car moving along a road that passes through some hills. Assume that there is an external observer who is recording the movement of the car using a pair of binoculars and a stopwatch. With a fast moving car, the observer must move the binoculars at sufficient speed to keep the car within sight. If the binoculars are moved too fast, the observer will view an area before the car has reached there; too slow, and the car will be out of sight because it is ahead of the viewed area. If the car changes speed or direction, the observer must adjust the movement of the binoculars to keep the car in view; if the car disappears behind a hill, the observer must use the car's recorded time, speed and direction to predict when and where it will re-emerge.

Suppose that the observer replaces the binoculars by an electronic camera which requires n seconds to process each frame and determine the position of the car. When the car is behind a hill, the observer must predict the position of the car and point the camera so that it keeps the car in the frame even though it is seen only at intervals of n seconds. To do this, the observer must model the movement of the car and, based on its past behaviour, predict its future movement. The observer may not have an explicit "model" of the car and may not even be conscious of doing the modelling; nevertheless, the accuracy of the prediction will depend on how faithfully the observer models the actual movement of the car.

Finally, assume that the car has no driver and is controlled by commands radioed by the observer. Being a physical system, the car will have some inertia and a reaction time, and the observer must use an even more precise model if the car is to be controlled successfully. Using information obtained every n seconds, the observer must send commands to adjust throttle settings and brake positions, and initiate changes of gear when needed. The difference between a driver in the car and the external observer, or remote controller, is that the driver has a continuous view of the terrain in front of the car and can adjust the controls continuously during its movement. The remote controller gets snapshots of the car every n seconds and must use these to plan changes of control.

1.1 Real-Time Computing

A real-time computer controlling a physical device or process has functions very similar to those of the observer controlling the car. Typically, sensors will provide readings at periodic intervals and the computer must respond by sending signals to actuators. There may be unexpected or irregular events and these must also receive a response. In all cases, there will be a time-bound within which the

response should be delivered. The ability of the computer to meet these demands depends on its capacity to perform the necessary computations in the given time.

If a number of events occur close together, the computer will need to schedule the computations so that each response is provided within the required time-bounds. It may be that, even so, the system is unable to meet all the possible demands and in this case we say that the system lacks sufficient resources (since a system with unlimited resources and capable of processing at infinite speed could satisfy any such timing constraint). Failure to meet the timing constraint for a response can have different consequences: in some cases, there may be no effect at all; in other cases, the effects may be minor and correctable; in yet other cases, the results may be catastrophic. Looking at the behaviour required of the observer allows us to define some of the properties needed for successful real-time control.

A real-time program must

- interact with an environment which has time-varying properties,
- exhibit predictable time-dependent behaviour, and
- execute on a system with limited resources.

Let us compare this description with that of the observer and the car. The movement of the car through the terrain certainly has time-varying properties (as must any movement). The observer must control this movement using information gathered by the electronic camera; if the car is to be steered safely through the terrain, responses must be sent to the car in time to alter the setting of its controls correctly. During normal operation, the observer can compute the position of the car and send control signals to the car at regular intervals.

If the terrain contains hazardous conditions, such as a flooded road or icy patches, the car may behave unexpectedly, for instance, skidding across the road in an arbitrary direction. If the observer is required to control the car under all conditions, it must be possible to react in time to such unexpected occurrences. When this is not possible, we can conclude that the real-time demands placed on the observer, under some conditions, may make it impossible to react in time to control the car safely. In order for a real-time system to manifest predictable time-dependent behaviour it is thus necessary for the environment to make predictable demands. With a human observer, the ability to react in time can be the result of skill, training, experience or just luck. How do we assess the real-time demands placed on a computer system and determine whether they will be met? If there is just one task and a single processor computer, calculating the real-time processing load may not be very difficult. As the number of tasks increases, it becomes more difficult to make precise predictions; if there is more than one processor, it is once again more difficult to obtain a definite prediction. There may be a number of factors that make it difficult to predict the timing of responses [39].

- A task may take different times under different conditions. For example, predicting the speed of a vehicle when it is moving on level ground can

be expected to take less time than if the terrain has a rough and irregular surface. If the system has many such tasks, the total load on the system at any time can be very difficult to calculate accurately.
- Tasks may have dependencies: Task A may need information from Task B before it can complete its calculation, and the time for completion of Task B may itself be variable. Under these conditions, it is only possible to set minimum and maximum bounds within which Task A will finish.
- With large and variable processing loads, it may be necessary to have more than one processor in the system. If tasks have dependencies, calculating task completion times on a multi-processor system is inherently more difficult than on a single processor system.
- The nature of the application may require distributed computing, with nodes connected by communication lines. The problem of finding completion times is then even more difficult, as communication between tasks can take varying times.

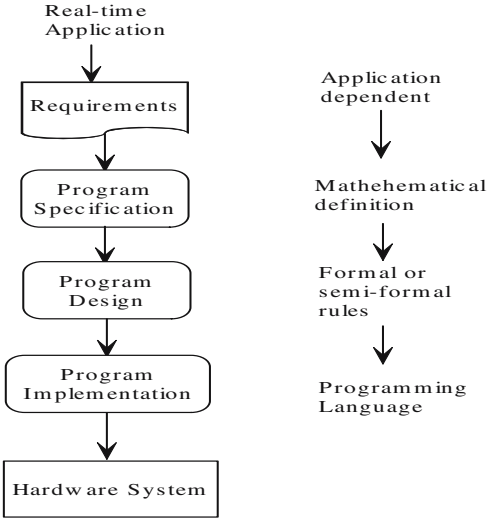


Fig. 1. Real-Time System Development

Requirements, Specification and Implementation. The demands placed on a real-time system arise from the needs of the application and are often called the requirements. Deciding on the precise requirements is a skilled task and can be carried out only with very good knowledge and experience of the application. Failures of large systems are often due to errors in defining the requirements. For a safety related real-time system, the operational requirements must then go through a hazard and risk analysis to determine the safety requirements. Requirements are often divided into two classes: functional requirements, which define the operations of the system and their effects and non-functional

requirements, such as timing properties. A system which produces a correctly calculated response but fails to meet its timing-bounds can have as dangerous an effect as one which produces a spurious result on time. So, for a real-time system, the functional and non-functional requirements must be precisely defined and together used to construct the specification of the system.

A specification is a mathematical statement of the properties to be exhibited by a system. A specification should be abstract so that

- it can be checked for conformity against the requirement, and
- its properties can be examined independently of the way in which it will be implemented as a program executing on a particular system.

This means that a specification should not enforce any decisions about the structure of the software, the programming language to be used or the kind of system on which the program is to be executed: these are properly implementation decisions. A specification is transformed into an application by taking design decisions, using formal or semi-formal rules, and converted into a program in some language (see Figure 1). We shall consider how a real-time system can be specified and implemented to meet the requirements. A notation will be used for the specification and it will be shown how the properties of the implementation can be checked. It will be noticed as the specifications unfold that there are many hidden complexities in even apparently simple real-time problems. This is why mathematical description and analysis have an important role to play, as they help to deal with this complexity. For both classical scheduling analysis and formal specification and verification in different notations, we refer the reader to [39].

1.2 An Example Real-Time System: Mine Pump

We illustrate the problem of real-time by a well-known case study [39]. Water percolating into a mine is collected in a sump to be pumped out of the mine (see Figure 2). The water level sensors D and E detect when water is above a high and a low level respectively. A pump controller switches the pump on when the water reaches the high water level and off when it goes below the low water level. If, due to a failure of the pump, the water cannot be pumped out, the mine must be evacuated within one hour.

The mine has other sensors (A, B, C) to monitor the carbon monoxide, methane and airflow levels. An alarm must be raised and the operator informed within one second of any of these levels becoming critical so that the mine can be evacuated within one hour. To avoid the risk of explosion, the pump must be operated only when the methane level is below a critical level.

Human operators can also control the operation of the pump, but within limits. An operator can switch the pump on or off if the water is between the low and high water levels. A special operator, the supervisor, can switch the pump on or off without this restriction. In all cases, the methane level must be below its critical level if the pump is to be operated.

Readings from all sensors, and a record of the operation of the pump, must be logged for later analysis.

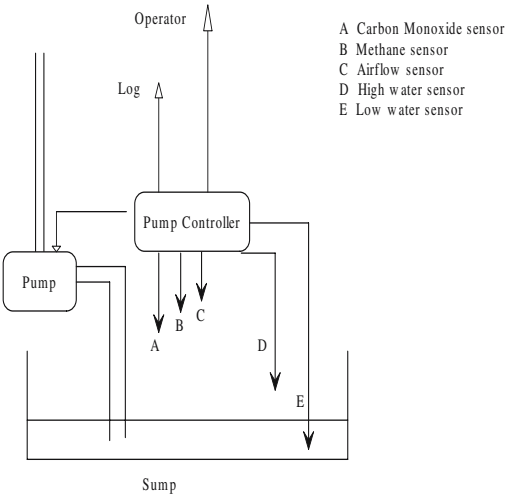


Fig. 2. Mine pump and control system (originally from Burns and Lister, 1991)

Safety Requirements. From the informal description of the mine pump and its operations we obtain the following safety requirements:

1. The pump must not be operated if the methane level is critical.
2. The mine must be evacuated within one hour of the pump failing.
3. Alarms must be raised if the methane level, the carbon monoxide level or the air-flow level is critical, in order for the be evacuated in time (say within one hour).

Operational Requirement. The mine is normally operated for three shifts a day, and the objective is for no more than one shift in *1000* to be lost due to high water levels.

Problem. Write and verify a specification for the mine pump controller under which it can be shown that the mine is operated whenever possible without violating the safety requirements.

Comments. The specification is to be the conjunction of two conditions: the mine must be operated when possible, and the safety requirements must not be violated. If the specification read "The mine must not be operated when the safety requirements are violated", then it could be trivially satisfied by not operating the mine at all! The specification must obviate this easy solution by requiring the mine to be operated when it is safely possible.

Note that the situation may not always be clearly defined and there may be times when it is difficult to determine whether operating the mine would violate the safety requirements. For example, the pump may fail when the water is at

any level; does the time of one hour for the evacuation of the mine apply to all possible water levels? More crucially, how is pump failure detected? Is pump failure always complete or can a pump fail partially and be able to displace only part of its normal output?

It is also important to consider under what conditions such a specification will be valid. If the methane or carbon monoxide levels can rise at an arbitrarily fast rate, there may not be time to evacuate the mine, or to switch off the pump. Unless there are bounds on the rate of change of different conditions, it will not be possible for the mine to be operated and meet the safety requirements. Sensors operate by sampling at periodic intervals and the pump will take some time to start and to stop. So the rate of change of a level must be small enough for conditions to not become dangerous during the reaction time of the equipment.

The control system obtains information about the level of water from the *Highwater* and *LowWater* sensors and of methane from the *Methane* sensor. Detailed data is needed about the rate at which water can enter the mine, and the frequency and duration of methane leaks; the correctness of the control software is predicated on the accuracy of this information. Can it also be assumed that the sensors always work correctly?

The description explains conditions under which the mine must be evacuated but does not indicate how often this may occur or how normal operation is resumed after an evacuation. For example, can a mine be evacuated more than once in a shift? After an evacuation, is the shift considered to be lost? If the mine is evacuated, it would be normal for a safety procedure to come into effect and for automatic and manual clearance to be needed before operation of the mine can resume. This information will make it possible to decide on how and when an alarm is reset once it has been raised.

1.3 Developing a Specification

We shall start by describing the requirements in terms of some properties, using a simple mathematical notation. This is a first step towards making a formal specification and we shall see various different, more complete, specifications of the problem in later chapters. Properties will be defined with simple predicate calculus expressions using the logical operators $\wedge$ (and), $\vee$ (or), $\Rightarrow$ (implies) and $\Leftrightarrow$ (iff), and the universal quantifier $\forall$ (for all). The usual mathematical relational operators will be used and functions, constants and variables will have types. We use

$$F : T_1 \rightarrow T_2$$

for a function F from type T_1 (the domain of the function) to type T_2 (the range of the function) and a variable V of type T will be defined as $V : T$. An interval from C_1 to C_2 will be represented as $[C_1, C_2]$ if the interval is closed and includes both C_1 and C_2, as $(C_1, C_2]$ if the interval is half-open and includes C_2 and not C_1 and as $[C_1, C_2)$ if the interval is half-open and includes C_1 and not C_2.

Assume that time is measured in seconds and recorded as a value in the set Time and the depth of the water is measured in metres and is a value in the set *Depth*; *Time* and *Depth* are the set of real numbers.

S1: Water level. The depth of the water in the sump depends on the rate at which water enters and leaves the sump and this will change over time. Let us define the water level *Water* at any time to be a function from *Time* to *Depth*:

$$Water : Time \rightarrow Depth$$

Let *Flow* be the rate of change of the depth of water measured in metres per second and be represented by a real number; *WaterIn* and *WaterOut* are the rates at which water enters and leaves the sump and, since these rates can change, they are functions from *Time* to *Flow*:

$$WaterIn,\ WaterOut : Time \rightarrow Flow$$

The depth of water in the sump at time t_2 is the sum of the depth of water at an earlier time t_1 and the difference between the amount of water that flows in and out in the time interval $[t1, t2]$. Thus for all $t1, t2 \in Time$ such that $t1 \leq t2$, we have

$$Water(t_2) = Water(t_1) + \int_{t_1}^{t_2} (WaterIn(t) - WaterOut(t))dt$$

HighWater and *LowWater* are constants representing the positions of the high and low water level sensors. For safe operation, the pump should be switched on when the water reaches the level *HighWater* and the level of water should always be kept below the level *DangerWater*:

$$DangerWater > HighWater > LowWater$$

If $HighWater = LowWater$, the high and low water sensors would effectively be reduced to one sensor.

S2: Methane level. The presence of methane is measured in units of pascals and recorded as a value of type *Pressure* (a real number). There is a critical level, *DangerMethane*, above which the presence of methane is dangerous.

The methane level is related to the flow of methane in and out of the mine. As for the water level, we define a function *Methane* for the methane level at any time and the functions *MethaneIn* and *MethaneOut* for the flow of methane in and out of the mine:

$$Methane : Time \rightarrow Pressure$$
$$MethaneIn, MethaneOut : Time \rightarrow Pressure$$

and for all $t_1, t_2 \in Time$,

$$Methane(t_2) = Methane(t_1) + \int_{t_1}^{t_2} (MethaneIn(t) - MethaneOut(t))dt$$

S3: Assumptions

1. There is a maximum rate $MaxWaterIn : Flow$ at which the water level in the sump can increase and at any time t, $WaterIn(t) \leq MaxWaterIn$.
2. The pump can remove water with a rate of at least $PumpRate : Flow$, and this must be greater than the maximum rate at which water can build up: $MaxWaterIn < PumpRate$.
3. The operation of the pump is represented by a predicate on Time which indicates when the pump is operating:

   $$Pumping : Time \rightarrow Bool$$

   and if the pump is operating at any time t it will produce an outflow of water of at least $PumpRate$:

   $$(Pumping(t) \wedge Water(t) > 0) \Rightarrow (WaterOut(t) > PumpRate)$$

4. There is enough reaction time t_P before the water level becomes dangerous;

   $$(HighWater + MaxWaterIn \cdot (t_P)) < DangerWater$$

5. The maximum rate at which methane can enter the mine is given by the constant $MaxMethaneRate$.
   If the methane sensor measures the methane level periodically every t_M units of time, and if the time for the pump to switch on or off is t_P, then the reaction time $t_M + t_P$ must be such that,

   $$(MaxMethaneRate \cdot t_m + HighMethane) < MethaneMargin \wedge$$
   $$(MaxMethaneRate \cdot t_P + MethaneMargin) < DangerMethane$$

   where $HighMethane < MethaneMargin < DangerMethane$. $HighMethane$ is the safety limit of methane and the methane is below this limit when the system starts. The controller should start to turn the pump off when it receives a methane level greater than $HighMethane$ signal from the sensor.
6. The methane level does not reach $MethaneMargin$ more than once in 1000 shifts; without this limit, it is not possible to meet the operational requirement. Methane is generated naturally during mining and is removed by ensuring a sufficient flow of fresh air, so this limit has some implications for the air circulation system.

S4: Pump controller. The pump controller must ensure that, under the assumptions, the operation of the pump will keep the water level within limits. At all times when the water level is high and the methane level is not critical, the pump is switched on, and if the methane level is critical the pump is switched off. Ignoring the reaction times, this can be specified as follows:

$$\forall\, t \in Time \cdot \begin{pmatrix} Water(t) > HighWater \wedge \\ Methane(t) < DangerMethane \end{pmatrix} \Rightarrow Pumping(t)$$

$$\wedge (Methane(t) \geq DangerMethane) \Rightarrow \neg Pumping(t))$$

This cannot really be achieved so let us see how reaction times can be taken into account. Since t_P is the time taken to switch the pump on, a properly operating controller must ensure that:

$$\forall\, t \in \mathit{Time} \cdot \begin{pmatrix} \mathit{Methane}(t) < \mathit{HighMethane} \wedge \neg \mathit{Pumping}(t) \wedge \\ \mathit{Water}(t) \geq \mathit{HighWater} \end{pmatrix} \\ \Rightarrow \exists\, t_0 \leq t_P \cdot \mathit{Pumping}(t + t_0)$$

So if the operator has not already switched the pump on, the pump controller must do so when the water level reaches *HighWater*. Similarly, the methane sensor may take t_M (later we assume $t_M = t_P$) units of time to detect a methane level and turn the pump off if the level is critical. That is, the pump controller must ensure that

$$\forall\, t \in \mathit{Time} \cdot \begin{pmatrix} \mathit{Pumping}(t) \wedge \\ \mathit{Methane}(t) \geq \mathit{HighMethane} \end{pmatrix} \Rightarrow \exists\, t_0 \leq t_M \cdot \neg \mathit{Pumping}(t + t_0)$$

S5: Sensors. Sensors are modelled by variables. The high water sensor provides information about the height of the water at time t in the form of predicates $HW(t)$ and $LW(t)$ which represent the cases where the water level is above *HighWater* and *LowWater* respectively. We assume that at all times a correctly working sensor gives some reading (that is, $HW(t) \vee \neg HW(t)$).

The readings provided by the sensors are related to the actual water level in the sump:

$$\forall\, t \in \mathit{Time} \cdot \mathit{Water}(t) \geq \mathit{HighWater} \Rightarrow HW(t) \\ \wedge\, \mathit{Water}(t) < \mathit{LowWater} \Rightarrow LW(t)$$

Note that when $\mathit{HighWater} = \mathit{LowWater}$, $LW(t) = \neg HW(t)$.

Similarly, the methane level sensor reads the methane level periodically and signals to the controller that either $\mathit{HML}(t)$ or $\neg \mathit{HML}(t)$:

$$\forall\, t \in \mathit{Time} \cdot \mathit{Methane}(t) \geq \mathit{HighMethane} \Rightarrow \mathit{HML}(t) \\ \wedge \mathit{Methane}(t) < \mathit{HighMethane} \Rightarrow \neg \mathit{HML}(t)$$

S6: Actuators. The pump is switched on and off by an actuator which receives signals from the pump controller. Once these signals are sent, the pump controller assumes that the pump acts accordingly. To validate this assumption, another condition is set by the operation of the pump. The outflow of water from the pump sets the condition *PumpOn*; similarly, when there is no outflow, the condition is *PumpOff*.

The assumption that the pump really is pumping when it is on and is not pumping when it is off is specified below: assume the pump takes κ time units to react after receiving the control signals of *PumpOn* and *PumpOff*:

$$\forall\, t \in \mathit{Time} \cdot \mathit{PumpOn}(t) \Rightarrow \exists\, t_0 \leq \kappa \cdot \mathit{Pumping}(t + t_0) \\ \mathit{PumpOff}(t) \Rightarrow \exists\, t_0 \leq \kappa \cdot \neg \mathit{Pumping}(t + t_0)$$

We can then refine the specification of the controller as

$$HW(t) \wedge LM(t) \Rightarrow \exists\, t_o \leq \varepsilon \cdot PumpOn(t + t_o)$$
$$HW(t) \wedge HM(t) \Rightarrow \exists\, t_o \leq \varepsilon \cdot PumpOff(t + t_o)$$

where ε is the time for the control program to produce the corresponding control command after receiving a sensor data, and $\varepsilon + \kappa \leq t_P$. Notice t_P is the total time needed in **S4** for the pump to be on or off when the water level and methane level require so.

The condition *PumpOn* is set by the actual outflow and there may be a delay before the outflow changes when the pump is switched on or off. If there were no delay, the implication $\Rightarrow$ could be replaced by the two-way implication *iff*, represented by $\Leftrightarrow$, and the two conditions *PumpOn* and *PumpOff* could be replaced by a single condition.

The *verification* of the system specification is about to prove

$$\begin{pmatrix}\text{(Controller Specification)} \wedge \\ \text{(Actuator Specification)} \wedge \\ \text{(Sensors Specification)}\end{pmatrix} \Rightarrow \begin{pmatrix}\text{Assumptions} \Rightarrow \\ \quad \text{(Requirement Specification)}\end{pmatrix}$$

where Controller Specification, Actuator Specification and Sensor Specification are respectively the conjunctions of the specifications in **S4**, **S5** and **S6**; Assumptions the conjunction of the assumptions specified in **S1**, **S2** and **S3**; and Requirement Specification is that was informally specified for the mine pump system (that can be formalized).

1.4 Constructing the Specification

The simple mathematical notation used so far provides a more abstract and a more precise description of the requirements than does the textual description. Having come so far, the next step should be to combine the definitions given in **S1**–**S6** and use this to prove the safety properties of the system. The combined definition should also be suitable for transformation into a program specification which can be used to develop a program.

Unfortunately, this is where the simplicity of the notation is a limitation. The definitions **S1**–**S6** can of course be made more detailed and perhaps taken a little further towards what could be a program specification. But the mathematical set theory used for the specification is both too rich and too complex to be useful in supporting program development. To develop a program, we need to consider several levels of specification (and so far we have just outlined the beginnings of one level) and each level must be shown to preserve the properties of the previous levels. This is the case for the controller specification in particular. The later levels must lead directly to a program and an implementation and there is nothing so far in the notation to suggest how this can be done. The interface of the control program and the physical environments must be also further specified.

What we need is a specification notation that has an underlying computational model which holds for all levels of specification. The notation must have a calculus or a proof system for reasoning about specifications and a method for transforming specifications to programs.

1.5 Analysis and Implementation

The development of a real-time program takes us part of the way towards an implementation. The next step is to analyze the timing properties of the program and, given the timing characteristics of the hardware system, to show that the implementation of the program will meet the timing constraints. It is not difficult to understand that for most time-critical systems, the speed of the processor is of great importance. But how exactly is processing speed related to the statements of the program and to timing deadlines?

A real-time system will usually have to meet many demands within limited time. The importance of the demands may vary with their nature (for instance, a safety-related demand may be more important than a simple data-logging demand) or with the time available for a response. The allocation of the resources of the system needs to be planned so that all demands are met by the time of their deadlines. This is usually done using a scheduler which implements a scheduling policy that determines how the resources of the system are allocated to the program. Scheduling policies can be analyzed mathematically so the precision of the formal specification and program development stages can be complemented by a mathematical timing analysis of the program properties. Taken together, specification, verification and timing analysis can provide accurate timing predictions for a real-time system.

We will discuss the relation between schedulability and verification and refinement.

2 Background and Overview

In this section we present an overview of formal techniques for real-time and fault tolerant systems. We also discuss how fault-tolerance and schedulability, as well as functional and time correctness, can be specified and verified within a single formal framework.

2.1 Historical Background of Formal Techniques in Real-Time and Fault-Tolerance

Starting from the early work in the 1970's, formal methods for concurrent and distributed systems development have seen considerable development. They have made a significant contribution to a better understanding of the behaviour of concurrent and distributed systems and to their correct and reliable implementation. The most widely studied methods include:

– Transition systems with temporal logic [214, 177, 110].
– Automata with Temporal logic [63, 37].
– Process algebras [115, 189].

Traditional temporal logic methods (and similar formalisms) use a *discrete event* approach: this is also the case with *transition systems* (such as [137, 214, 176, 142, 215]), *automata* [63, 61] and *action systems* [15, 53]). Such models abstract away time in the behaviour and describe the ordering of the events of a system.

Real-time is introduced into transition systems either by associating lower and upper bounds with enabled transitions [216] or by introducing explicit clocks [9, 2]. For specification and verification, a temporal logic is then extended either with the introduction of bounded (or *quantized*) temporal operators [216, 139] or with the addition of explicit clock variables [216, 7, 2]. The relationship between the two approaches, the extent to which one can be translated into another, is investigated in [110].

One approach to the construction of safe and dependable computing systems is to use formal specification, development and verification methods as part of a *fault-intolerance approach* in which the system safety and dependability are improved by *a priori fault avoidance* and *fault removal* [13]. Another path towards this goal is through *fault-tolerance*, which is complementary to fault-intolerance but not a substitute for it [13]. This is based on the use of protective redundancy: a system is designed to be *fault-tolerant* by incorporating additional components and algorithms to ensure that the occurrence of an error state does not result in later system failures [221, 222, 147]. Although fault-tolerance is by no means a new concept [202], there was little work on formal treatment of fault-tolerance until the 1980s [191, 232, 76, 159, 162, 163, 164]. These papers treat untimed fault-tolerant systems only. Recent work [70, 231, 146, 165] has shown how fault-tolerance and timing properties can formally be treated uniformly.

The issue of schedulability arises when a real-time program is to be implemented on a system with limited resources (such as processors) [135]. An infeasible implementation of a real-time program will not meet the timing requirement even though the program has been formally proven correct. Schedulability has been for a long time a concern of scheduling theory [158, 136, 150, 12, 40] but the models and techniques used there are quite different from those used in formal specification and development methods. The relationship between the computational model used in a scheduling analysis and the model (such as an interleaving model) used in a formal development is not clear. Thus, results obtained in scheduling theory are hard to relate to or use in the formal development of a system. It is however possible to verify the schedulability of a program within a formal framework [212, 261, 93, 169, 173] and this provides a starting point for a proof-theoretic interpretation of results from scheduling theory.

2.2 Overview of the Formal Framework

We now show how fault-tolerance and schedulability, as well as functional and time correctness, can be specified and verified within a single formal framework.

We use *transition systems* [137, 214] as the program model, and the Temporal Logic of Actions (TLA) [144, 145] as the specification notation. Physical faults in a system are modelled as being caused by a set F of *fault actions* which perform state transformations in the same way as other program actions. Fault-tolerance is achieved if the program can be made tolerant to these faults (for instance, by adding the appropriate recovery actions [221, 222, 159, 162, 163, 164]). We shall show that proof of fault-tolerance is no different to proof of any functional property.

Each action τ of a real-time program is associated with a *volatile lower bound* $L(\tau)$ and a *volatile upper bound* $U(\tau)$, meaning that action τ can be performed only if it has been *continuously* enabled for at least $L(\tau)$ time units, and τ must not be *continuously* enabled for $U(\tau)$ time units without being performed. The use of volatile time bounds or, correspondingly, *volatile timers* (or *clock variables*) in the explicit-clock modelling approach has been described in the literature (see the references in the previous subsection) to specify the time-criticality of an operation.

To deal with real-time scheduling, it is important to model actions and their pre-emption at a level of abstraction suitable for measuring time intervals and to ensure that pre-emption of an execution respects the atomicity of actions. To achieve this, we use *persistent time bounds* [166] to constrain the *cumulative execution time* of an action in the execution of a program under a scheduler. The persistent lower bound $l(\tau)$ for an action τ means that action τ can be performed (or finished) only if it has been executed by a processor for at least a *total* of $l(\tau)$ time units, not necessarily continuously'; the persistent upper bound $u(\tau)$ means that 'τ is not executed by a processor for a *total* of $u(\tau)$ time units without being completed.

In TLA, programs and properties are specified as logical formulas, and this allows the logical characterisation and treatment of the *refinement relation* between programs. We shall show how, using this approach, the untimed program, the fault assumptions, the timing assumptions and scheduling policies are specified as separate TLA formulas.

The use of a well established computational model and logic has significant advantages over the use of a specially designed semantic model and logic (as in [232, 76, 70, 231] for fault-tolerance and [212, 173, 91] for schedulability). First, less effort is needed to understand the model and the logic. Second, existing methods for specification, refinement and verification can be readily applied to deal with fault-tolerance and schedulability. Also, existing mechanical proof assistance (such as [86, 29] and model-checking methods and tools (such as [63, 8, 112]) can be used [9, 7, 37].

3 Program Specification, Verification and Refinement

This section introduces a transition system which is widely used as the computational model in temporal logic. This model serves as the semantic model of TLA that we shall use for specification and verification.

3.1 Introducing TLA

Values, variables and states. TLA is a logic used for specifying and reasoning about programs which manipulate data. Assume there is a set *Val* of *values*, where a value is a data item. We assume that *Val* contains all the values, such as numbers like *3*, strings such as *abc* and sets like *Nat*, needed for our programs.

Assume that a program manipulates data by changing its *state*, which is an assignment of values to *state variables*. For describing all possible programs, we assume an *infinite* set *Var* of variables, which are represented by symbols like x, y, z. A *state* s is thus a mapping from *Var* to *Val*:

$$s : Var \longmapsto Val$$

For a state s, the value assigned to a variable x in state s is represented by $s[x]$ and the values assigned to a subset $\overline{z}$ of variables is denoted by $s[\overline{z}]$. Given a subset $\overline{v} \subseteq Val$ of variables, we also define *a state* s *over* $\overline{v}$ to be a mapping from $\overline{v}$ to *Val*.

Examples of States. For state variables: $\{x, y\}$, let

- $s = \{x \mapsto 0, y \mapsto 1\}$, $s' = \{x \mapsto 1, y \mapsto 0\}$
- $s[x] = 0$, $s[y] = 1$, $s'[x] = 1$, $s'[y] = 0$

Assume $\{On, Off, Bright\}$ are state variables used to model a light system:

- $s_1 = \{\ Off \mapsto true, On \mapsto false, Bright \mapsto false\ \}$
  $s_2 = \{\ Off \mapsto false, On \mapsto true, Bright \mapsto false\ \}$
  $s_3 = \{\ Off \mapsto false, On \mapsto true, Bright \mapsto true\ \}$
- $s_1[On] = false$, $s_2[On] = true$, etc.

State predicates. A *state predicate*, called a *predicate* for short, is a first-order Boolean-valued expression built from variables and constant symbols. For example, $(x = y - 3) \wedge x \in Nat$. The meaning $[\![Q]\!]$ of a predicate is a mapping from states to Booleans $\{true, false\}$ once an *interpretation* is given to the predicate symbols (like $=$) and the function symbols (like $-$) used in Q. We say that a state s *satisfies* a predicate Q, denoted by $s \models Q$, iff $[\![Q]\!](s) = true$.

Consider variables $\{x, y\}$ and initial and final states $s = \{\ x \mapsto 0, y \mapsto 1\ \}$ and $s' = \{x \mapsto 1, y \mapsto 0\}$. Then $(x-1 = y)$, $(x+y > 3)$ and $(x-1 = y) \vee (x+y > 3)$ are all predicates. Assume x and y take values from the integers and the meanings of the equality symbol $(=)$, inequality symbol $(>)$ and the function symbols $(-$ and $+)$ are those defined in the arithmetic on integers. We can easily decide which of the predicates are satisfied by state s and which by s'.

Actions. The execution of a program changes the state of the program by the execution of *atomic actions*, called an *actions* for short. An *action* is a first-order Boolean-valued expression over the variables *Var* and their primed versions *Var*$'$. For example, $x' + 1 = y$ and $x' \geq y' + (x - 1)$ are actions.

For a given interpretation of the predicate symbols (such as $=$ and $\geq$) and an interpretation of the function symbols (such as $+$ and $-$), an action defines a relation between the *values* of variables before and the *values* of primed variables after the execution of the action. Formally, given the interpretation of the predicate and function symbols, the meaning $[\![\tau]\!]$ of an action τ is a relation between states, that is, a function that assigns a Boolean value to a pair (s, s') of states. We thus define $[\![\tau]\!](s, s')$ by considering s to be the *pre-τ-state* and s' the *post-τ-state* and $[\![\tau]\!](s, s')$ is obtained from τ by replacing each unprimed variable x it τ by its value $s[x]$ in s and each primed variable x' in τ by the value $s'[x]$ of x in s:

$$[\![\tau]\!](s, s') = \mathit{true} \text{ iff } \tau(s[\overline{z}]/\overline{z}, s'[\overline{z}]/\overline{z}') \text{ holds}$$

where $\overline{z}$ and $\overline{z}'$ are the sets of unprimed and primed variables in τ. We say that a pair (s, s') of states *satisfies* an action τ, denoted by $(s, s') \models \tau$, if, and only if, $[\![\tau]\!](s, s') = \mathit{true}$. When $(s, s') \models \tau$, (s, s') is called a *τ-step*.

A predicate Q can also be viewed as a particular action which does not have primed variables. Thus Q is satisfied by a pair (s, s') of states iff it is satisfied by the first state s in the pair. For an action τ, let $en(\tau)$ be the predicate, called the *enabling condition* (or *guard*) of τ, which is true of a state s iff there exists a state s' such that $(s, s') \models \tau$. Formally, let $x'_1, \ldots, x'_n$ be the primed variables that occur in τ, let $\hat{x}_1, \ldots, \hat{x}_n$ be new logical variables that do not occur in τ. The enabling condition is the predicate defined below

$$en(\tau) \stackrel{\Delta}{=} \exists\, \hat{x}_1, \ldots \hat{x}_n \cdot \tau[\hat{x}_1, \ldots, \hat{x}_n / x'_1, \ldots, x'_n]$$

where $\tau[\hat{x}_1, \ldots, \hat{x}_n / x'_1, \ldots, x'_n]$ is the formula obtained from τ by replacing each occurrence of x'_i by $\hat{x}_i$, for $i = 1, \ldots, n$. Note that $en(\tau)$ does not contain primed variables. For example, assume that x takes values from integers. the enabling condition of τ be $(x > 0) \wedge (x' = x - 1)$ is

$$\begin{aligned} en(\tau) &= \exists\, \hat{x} \cdot ((x > 0) \wedge (\hat{x} = x - 1)) \\ &= x > 0 \end{aligned}$$

Temporal formulas. We consider an execution of a program to be an *infinite* state sequence, and take the semantics of the program to be the set of all its possible executions. Reasoning about programs is reasoning about their executions and thus reasoning about state sequences. We shall use TLA for this purpose.

Formulas in TLA are called *temporal formulas* which are built from actions as the *elementary temporal formulas* using Boolean connectives and modal operators in Linear-Time Temporal Logic [177]. Here we use only $\Box$ (read *always*) and its dual operator $\Diamond$ (read *eventually*) defined as $\neg\Box\neg$. Quantification (that is, $\exists\,\overline{x}$, $\forall\,\overline{x}$) is possible over a set of *logical* (or *rigid*) variables, whose values are fixed over states, and over a set of state variables, whose values can change from state to state. Please note that in n [144, 145], the bold versions $\exists$ and $\forall$ are used to quantify state variables.

To use these formulas for describing state sequences, it requires to define the semantic meaning of such a formula as a function from executions to Booleans. We must first lift the semantics of an action based on pairs of states to one based on state sequences.

Given an infinite sequence $\sigma = \sigma_0, \sigma_1, \ldots$ of states, then:

- An action $[\![\tau]\!](\sigma) = \mathit{true}$ iff $[\![\tau]\!](\sigma_0, \sigma_1) = \mathit{true}$. Note that $[\![\tau]\!]$ is overloaded here.
- The first-order connectives and quantification over logical variables retain their standard semantics.
- $[\![\Box\varphi]\!](\sigma) = \mathit{true}$ iff $[\![\varphi]\!](\eta) = \mathit{true}$ for any suffix η of σ. This implies that $[\![\Diamond\varphi]\!](\sigma) = \mathit{true}$ iff $[\![\varphi]\!](\eta) = \mathit{true}$ for some suffix η of σ.
- $[\![\exists\, x \cdot \varphi]\!](\sigma) = \mathit{true}$ iff there is an η such that $\sigma =_x \eta$ and $[\![\varphi]\!](\eta)$, where the relation $\sigma =_x \eta$ holds between state sequences σ and η iff $\sigma_i[y] = \eta_i[y]$ for any variable y which differs from x and for any $i \geq 0$. Thus, $\exists\, x \cdot \varphi$ is true of σ iff φ is true of some infinite state sequence η that differs from σ only in the values assigned to the variable x.

We say that a formula φ is satisfied by σ, denoted by $\sigma \models \varphi$, if $[\![\varphi]\!](\sigma)$.

Exercise 1. For state variables $\{x, y\}$ and a state sequence:

$$\sigma = s_1, s_2, s_1, s_2, \ldots$$

where $s_1 = \{\, x \mapsto 0, y \mapsto 1 \}$ and $s_2 = \{x \mapsto 1, y \mapsto 0\,\}$, which of the following relations hold?

1. $\sigma \models (x = y - 1)$
2. $\sigma \models \Box(x + 1 = 1)$
3. $\sigma \models \Box(y + y' = 1)$
4. $\sigma \models \exists\, x \cdot \Box((y \geq 0) \wedge (x = 1))$
5. $\sigma \models \Box\Diamond(y' = y - 1)$
6. $\sigma \models \Box\Diamond(y' = y + 1)$
7. $\sigma \models \Box(\Diamond(y' = y - 1) \wedge \Diamond(y' = y + 1))$

A formula φ is *valid* if it is satisfied by any infinite state sequences over *Var*. A relatively complete proof system is given in [144], with additional rules for using the logic for reasoning about programs. Every valid TLA formula is provable from the axioms and proof rules of the TLA proof system if all the valid action formulas are provable. As the temporal operators $\Box$ and $\Diamond$ and the semantic model are the same as those in [177], the rules and methods provided there for verification can also be used.

Exercise 2. Which of the following formulas are valid?

- $\Box(\varphi \vee \neg\varphi)$, $\Box\varphi \Rightarrow \varphi$, $\varphi \Rightarrow \Diamond\varphi$, $\Box\varphi \Rightarrow \Diamond\varphi$, $\Diamond(\varphi \vee \psi) \Rightarrow \Diamond\varphi \vee \Diamond\psi$
- $\Diamond(\varphi \wedge \psi) \Rightarrow \Diamond\varphi \wedge \Diamond\psi$, $\Box(\varphi \vee \psi) \Rightarrow \Box\varphi \vee \Box\psi$, $\Box(\varphi \wedge \psi) \Rightarrow \Box\varphi \wedge \Box\psi$

3.2 The Computational Model and Program Specification

We now give a mathematical definition of a program.

Definition 1. *A* **program** *will be represented as an* action system *(or* a transition system*) which is a tuple* $P = (\overline{v}, \overline{x}, \Theta, A)$ *consisting of four components:*

1. *A finite non-empty set* $\overline{v}$ *of* state variables.
2. *A set* $\overline{x}$ *of* internal variables*, which is a subset of* $\overline{v}$ *and possibly empty. The values of these internal variables are not observable to the environment of the program.*
3. *An* initial condition Θ *which is a state predicate referring to only variables in* $\overline{v}$ *that defines the set of* initial states *of the program.*
4. *A finite set* A *of* atomic actions *in which only variables in* $\overline{v}$ *and primed variables in* $\overline{v}'$ *can occur.*

The simple light control system can also be modelled as

- $\overline{v} \triangleq \{s\}$, $\overline{x} \triangleq \{\ \}$
- $\Theta \triangleq (s = \mathit{off}$
- $A \triangleq \left\{ \begin{matrix} a : (s = \mathit{off}) \wedge (s' = \mathit{on}), & b : (s = \mathit{on}) \wedge (s' = \mathit{off}), \\ c : (s = \mathit{on}) \wedge (s' = \mathit{bright}), & d : (s = \mathit{bright}) \wedge (s' = \mathit{off}) \end{matrix} \right\}$

Notice that this model is not quite precise yet as it does not say what cannot be changed by an action. One can imagine think of combining action b and c by a disjunction $\vee$, that will illustrate the nondeterminism of a system.

Consider the composition of the two open systems:

- Light control system: *LightC*:

$$\begin{aligned} &(s = \mathit{off} \wedge \mathit{button} = \mathit{pressed}) \wedge (s' = \mathit{on} \wedge \mathit{button}' = \mathit{released}), \\ &(s = \mathit{on} \wedge \mathit{button} = \mathit{pressed}) \wedge (s' = \mathit{off} \wedge \mathit{button}' = \mathit{released}), \\ &(s = \mathit{on} \wedge \mathit{button} = \mathit{pressed}) \wedge (s' = \mathit{bright} \wedge \mathit{button}' = \mathit{released}), \\ &(s = \mathit{bright} \wedge \mathit{button} = \mathit{pressed}) \wedge (s' = \mathit{off} \wedge \mathit{button}' = \mathit{released}) \end{aligned}$$

- The Button: *Button* with one action:

$$(\mathit{button} = \mathit{released}) \wedge (\mathit{button}' = \mathit{pressed})$$

LightC $\parallel$ *Button* has the union of the set of variables and the union of the sets of actions of the two components.

Definition 2. *A* **computation** *(also called an execution or a run) of a program* $P = (\overline{v}, \overline{x}, \Theta, A)$ *is an infinite sequence* $\sigma = \sigma_0, \sigma_1, \ldots$ *over* $\overline{v}$ *such that the following two conditions hold:*

Initiality*:* σ_0 *satisfies* Θ.

Consecution*: For all* $i \geq 0$*, either* $\sigma_i = \sigma_{i+1}$ *(a* stuttering step*) or there is an action* τ *in* A *such that* (σ_i, σ_{i+1}) *is a* τ*-step (a* diligent step*). In the latter case, we say that a* τ *step is* taken *at position* i *of* σ.

Thus a computation either contains infinitely many diligent steps, or a diligent step takes it to a terminating state after which only stuttering steps occur; in this case we say that the computation is *terminating*.

The set of all the computations of a program is *stuttering closed*: if an infinite state sequence σ is a computation of the program, then so is any state sequence obtained from σ by adding or deleting a finite number of stuttering steps.

Remarks on atomicity, interleaving and concurrency

- Atomicity is a means of modelling *mutual exclusion synchronization*.
- Guarded atomic actions are for *conditional synchronization*. Notice that the guard of an action here is the enabling condition of the action.
- A number of atomic actions can be executed *in parallel iff* the order in which they take place does not affect the changes in the states. In this model, the use of an interleaving semantics works well for concurrent systems.
- An atomic action can be understood, and in fact *often though not always* implemented, as a piece of a sequential *terminating* program.
- A piece of a terminating concurrent program (nested parallelism) is equivalent to a non-deterministic sequential program (this is also captured by the expansion law of CCS [189] and CSP [115]).

Notice that in this definition, an atomic action in a program is semantically taken just as a binary relation on the states. Therefore, although the actions in the set A are syntactically distinct from each other, we do not require that the actions be mutually disjoint in their semantics; in particular, one action can semantically be a sub-relation of another. This implies that it is possible that two actions have the same effect on a single state. This does not cause any theoretical problem, as we are to reason about properties of the execution of the program, not the effect of a individual action. In practice, when we use this model to define the semantics of a concurrent program, each atomic action defines a different piece of code of the program. Then the effect of all the actions obtained from the program will be different in any state, as they at least modify different control variables, such as process counter variables, which are usually internal variables.

An atomic action of a program usually changes only a subset of the variables of the program, leaving the others unchanged. For a finite set $\overline{z}$ of variables, we define:

$$\mathit{unchanged}(\overline{z}) = \bigwedge_{x \in \overline{z}} (x' = x)$$

For example, the atomic action $x > 0 \longrightarrow x := x - 1$ (in the form of a guarded command) can be described as the action formula:

$$(x > 0) \wedge (x' = x - 1) \wedge \mathit{unchanged}(\overline{v} - \{x\})$$

in which $x > 0$ is the enabling condition (or the guard).

In the examples, we will simply omit the *unchanged* part when we specify an action, by assuming it changes the values of only those variables whose primed versions are referred to in the action formula.

To specify stuttering, we define also an abbreviation for an action τ and a finite set of state variables $\overline{z}$:

$$[\tau]_{\overline{z}} \stackrel{\Delta}{=} \tau \vee \mathit{unchanged}(\overline{z})$$

asserting that a step is either a τ-step or a step which does not change the values of the state variables $\overline{z}$.

We are ready to define two normal forms of program specifications.

Definition 3. *Given a program* $P = (\overline{v}, \overline{x}, \Theta, A)$*, let:*

$$\mathcal{N}_P \stackrel{\Delta}{=} \bigvee_{\tau \in A} \tau$$

$\mathcal{N}_P$ *is the* state-transition relation *for the atomic actions of* P*. The* **exact** *(or* **internal***)* specification *of* P *is expressed by the formula:*

$$\Pi(P) \stackrel{\Delta}{=} \Theta \wedge \Box[\mathcal{N}_P]_{\overline{v}}$$

An exact specification defines all the possible sequences of values that may be taken by the state variables, including the *internal* variables $\overline{x}$. Existential quantification can be used to hide the internal variables $\overline{x}$ which automatically get their adequate values although they are not visible to the observer.

Definition 4. *The* **canonical** *(or* **external***)* safety specification *of* P *is given as:*

$$\Phi(P) \stackrel{\Delta}{=} \exists\, \overline{x} \cdot \Pi(P)$$

An infinite state sequence σ over $\overline{v}$ satisfies $\Phi(P)$ iff there is an infinite state sequence η that satisfies $\Pi(P)$ and differs from σ only in the values assigned to the variables x_i, $i = 1, \ldots, n$.

Importance of the stuttering closure property. Stuttering closure is important when using a specification of a system in a larger system. In that case, the actions of this subsystem will be interleaved with other actions in the larger system, and the variables of the subsystem will not be changed when actions of the rest of the system take place.

To understand this point, consider a digital clock that displays only the hour. Let *hr* represents the clock's display.

From any starting hour, say *11*, the behaviour of the clock is trivially:

$$\{\, hr \mapsto 11 \,\} \longrightarrow \{hr \mapsto 12\} \longrightarrow \{hr \mapsto 1\} \longrightarrow \{hr \mapsto 2\,\} \cdots$$

Each step is carried out by the action

$$\mathit{HCnext} \stackrel{\Delta}{=} hr' = (hr \bmod 12) + 1$$

The clock can be specified by $HCinit \wedge \Box HCnext$, where $HCinit$ is the initial condition that the clock starts from any hour:

$$HCinit \triangleq hr \in \mathbb{N} \wedge (1 \leq hr \leq 12)$$

This will work if the clock is considered in isolation and never related to another system. However, this specification cannot be re-used when we model a device that displays the current hour and temperature.

$$\left\{\begin{array}{l} hr \mapsto 11, \\ temp \mapsto 3.5 \end{array}\right\} \longrightarrow \left\{\begin{array}{l} hr \mapsto 12, \\ temp \mapsto 3.5 \end{array}\right\} \longrightarrow \left\{\begin{array}{l} hr \mapsto 12, \\ temp \mapsto 3 \end{array}\right\} \longrightarrow \left\{\begin{array}{l} hr \mapsto 12, \\ temp \mapsto 2.5 \end{array}\right\} \ldots$$

Therefore, stuttering is essential for composition. We will also see later that the stuttering closure property is the key to deal with refinement between two programs.

Exercise 3. Write a specification *Tempclock* for a digital clock that displays the current hour and temperature such that:

$$TempClock = HClock \wedge TempDisplay$$

where

$$HClock \triangleq HCinit \wedge [HCnext]_{hr}$$

Formulas $\Pi(P)$ and $\Phi(P)$ are safety properties and they are satisfied by an infinite state sequence iff they are satisfied by every finite prefix of the sequence. Safety properties allow computations in which a system performs correctly for a while and then leaves the values of all variables unchanged.

For an action τ, define the action $\langle\tau\rangle_{\overline{z}} \triangleq \tau \wedge \neg unchanged(\overline{z})$. Then we can specify the following fairness properties.

$$\textit{Weak fairness: } WF_{\overline{z}}(\tau) \triangleq (\Box\Diamond\langle\tau\rangle_{\overline{z}}) \vee (\Box\Diamond\neg en(\langle\tau\rangle_{\overline{z}}))$$
$$\textit{Strong fairness:} SF_{\overline{z}}(\tau) \triangleq (\Box\Diamond\langle\tau\rangle_{\overline{z}}) \vee (\Diamond\Box\neg en(\langle\tau\rangle_{\overline{z}}))$$

The weak fairness condition $WF_{\overline{z}}(\tau)$ says that from any point in an execution, the action τ must eventually be performed if it remains enabled until it is performed. The strong fairness condition $SF_{\overline{z}}(\tau)$ says that from any point in an execution, the action τ must be eventually executed infinitely often if it is infinitely often enabled.

The safety specifications $\Pi(P)$ and $\Phi(P)$ are usually strengthened by conjoining them with one or more fairness properties: $\Pi(P) \wedge \mathcal{L}$ and $\exists\,\overline{x} \cdot (\Pi(P) \wedge \mathcal{L})$, where $\mathcal{L}$ is the specification of the fairness conditions.

3.3 Running Example

The processor of a simple system issues *read* and *write* operations to be executed by the memory. The processor-memory interface has two registers, represented by the following state variables:

op: Set by the processor to the chosen operation, and reset by the memory after execution; its value space is $\{rdy, r, w\}$, for *ready*, *read* and *write*, respectively.

val: Set by the processor to the value v to be written by a *write*, and by the memory to return the result of a *read*; its value space is the set of integers, $\mathbf{Z}$.

Let the interface be a program P_1 with an (*internal*) variable d when denotes the data in the memory.

$$
\begin{array}{rll}
\overline{v}_1 \triangleq & \{op, val, d\} & \\
\Theta_1 \triangleq & (op \in \{rdy, r, w\}) \wedge d \in \mathbf{Z} & \text{initial condition} \\
R_1^p \triangleq & (op = rdy) \wedge (op' = r) & \text{processor issues } \textit{read} \\
W_1^p \triangleq & (op = rdy) \wedge (op' = w) \wedge (val' \in \mathbf{Z}) & \text{processor issues } \textit{write} \\
R_1^m \triangleq & (op = r) \wedge (op' = rdy) \wedge (val' = d) & \text{memory executes } \textit{read} \\
W_1^m \triangleq & (op = w) \wedge (op' = rdy) \wedge (d' = val) & \text{memory executes } \textit{write}
\end{array}
$$

$$
\begin{array}{rll}
A_1 = & \{ R_1^p, W_1^p, R_1^m, W_1^m \} & \text{actions of the program} \\
P_1 = & (\overline{v}_1, \Theta_1, A_1) & \text{the program} \\
\mathcal{N}_{P_1} = & R_1^p \vee W_1^p \vee R_1^m \vee W_1^m & \text{state transition relation} \\
\Pi(P_1) = & \Theta_1 \wedge \Box[\mathcal{N}_{P_1}]_{\overline{v}_1} & \text{exact specification} \\
\Phi(P_1) = & \exists\, d \cdot \Theta_1 \wedge \Box[\mathcal{N}_{P_1}]_{\overline{v}_1} & \text{hiding the internal variable } d
\end{array}
$$

The two actions R_1^p and W_1^p of the processor can be combined into a single nondeterministic action:

$$RW_1^p \triangleq (op = rdy) \wedge ((op' = r) \vee (op' = w) \wedge (val' \in \mathbf{Z}))$$

3.4 Verification and Refinement

In TLA, verification of a program property specified by a formula φ which does not contain free internal variables is by proving the validity of the implication $\Phi(P) \Rightarrow \varphi$. A relatively complete proof system is given in [144], with additional rules for using the logic for reasoning about programs. Every valid TLA formula is provable from the axioms and proof rules of the TLA proof system if all the valid action formulas are provable. As the temporal operators $\Box$ and $\Diamond$ and the computational model are the same as those in [177], the rules and methods provided there for verification can be used.

Definition 5. *The relation $P_h \sqsubseteq P_l$ between two programs $P_h = (\overline{v}_h, \overline{y}, \Theta_h, A_h)$ and $P_l = (\overline{v}_l, \overline{x}, \Theta_l, A_l)$ characterizes* **refinement** *and can be understood as that program P_h is correctly implemented by P_l. Let*

$$
\begin{array}{l}
\Phi(P_h) = \exists\, \overline{y} \cdot \Theta_h \wedge \Box[\mathcal{N}_{P_h}]_{\overline{v}_h} \\
\Phi(P_l) = \exists\, \overline{x} \cdot \Theta_l \wedge \Box[\mathcal{N}_{P_l}]_{\overline{v}_l}
\end{array}
$$

be canonical specifications of P_l and P_h respectively, where

$$\overline{x} = \{x_1, \ldots, x_n\} \qquad \overline{y} = \{y_1, \ldots, y_m\}$$

Then the refinement relation *is formalized as:*

$$P_h \sqsubseteq P_l \quad \text{iff} \quad \Phi(P_l) \Rightarrow \Phi(P_h)$$

To prove the implication, we must define state functions $\tilde{y}_1, \ldots, \tilde{y}_m$ in terms of the variables $\overline{v}_l$ and prove the implication $\Pi(P_l) \Rightarrow \widetilde{\Pi(P_h)}$, where $\widetilde{\Pi(P_h)}$ is obtained from $\Pi(P_h)$ by substituting $\tilde{y}_i$ for all the free occurrences of y_i in $\Pi(P_h)$, for $i = 1, \ldots, m$. The collection of state functions $\tilde{y}_1, \ldots, \tilde{y}_m$ is called a *refinement mapping*. The substitutions can be applied also to a sub-formula of $\Pi(P_h)$. $\tilde{y}_i$ is the *concrete* state function with which P_l implements the *abstract* variable y_i of P_h. The proof of the implication can be carried out in two steps:

1. *initiality-preservation:* $\Theta_l \Rightarrow \widetilde{\Theta}_h$;
2. *step-simulation:* $\mathcal{N}_{P_l} \Rightarrow [\widetilde{\mathcal{N}}_{P_h}]_{\widetilde{\overline{v}}_l}$.

As $\mathcal{N}_{P_l}$ is the disjunction of the actions of P_l, step-simulation can be proved by showing $\tau \Rightarrow [\widetilde{\mathcal{N}}_{P_h}]_{\widetilde{\overline{v}}_l}$ for each $\tau \in A_l$; each step of the state transition by P_l corresponds to either a diligent step or a stuttering step by P_h.

Completeness Remarks. The validity of the implication $\Phi(P_l) \Rightarrow \Phi(P_h)$ does not imply the existence of a refinement mapping, but in general, refinement mappings, can be found by adding dummy (or auxiliary) variables to specifications [1]. Once a refinement mapping is found, the verification of the refinement is straightforward and can be aided by mechanical means (such as [86]). However, finding a refinement mapping may be difficult if it is not known how P_l is obtained from P_h. On the other hand, knowing how an abstract state variable in P_h is implemented by the variables in P_l, it is possible to define the mapping between them. Refinement supports stepwise development in which a small number of abstract state variables are refined in each step.

Exercise 4. Consider the problem of Dinning Philosophers. Assume there are five philosophers, p_i, $i = 1, \ldots, 5$, and five chopsticks, c_i, $i = 1, \ldots, 5$, that are placed in five positions a dinning table. The life of each philosopher is repeatedly *thinking* and *eating*. Assume that initially, all philosophers are thinking. After thinking, a philosopher p_i becomes hungry and want to eat. To eat, he has to come to the position represented by the chair at the dinning table reserved form him and gets the chopstick c_i on his left and then the one, c_{i+1}, on his right. A philosopher cannot start to eat before he gets both sticks. After eating, a philosopher puts down both chopsticks and goes back to think.

1. Write a TLA specification of the problem of the dinning philosophers.
2. Specify in TLA that the fairness condition that an eating philosopher will eventually put down the chopsticks.

3. Specify the liveness property that no philosopher can be starved.
4. Does your specification for part (1) satisfies the liveness property under the fairness condition (deadlock freedom)?
5. Suggest solutions to fix deadlock problem in the specification of part (1), and write the TLA specifications for these solutions.

3.5 Linking Theories of Programming to TLA

The use of atomic actions allows us to use most of the theories of sequential programming smoothly in this framework. In concurrent programming, an atomic action is often implemented as a *guarded command* which can be a big piece of program text [81] (also see Morgan's chapter on 'Probability in the context of wp' in this volume). A guarded command is of the form $g \longrightarrow C$, where C can be any programming statement such as

$$
\begin{array}{lll}
C ::= & x := e & \\
| & C;\ C & \text{sequential composition} \\
| & C \sqcap C & \text{nondeterministic choice} \\
| & C \lhd b \rhd C & \text{conditional choice} \\
| & b * C & \text{iteration}
\end{array}
$$

For a given a command C, we can calculate a design $[\![C]\!]$ following the calculus of UTP [117]:

$$Pre_C \vdash Post_C$$

The corresponding TLA action of $g \longrightarrow C$ is then

$$g \wedge (Pre_C \Rightarrow Post_C)$$

We can use $(Pre_C \Rightarrow Post_C) \lhd g \rhd Skip$, as we allow stuttering.

Also, reasoning about TLA specifications, such as verifying an invariant property $\Box Q$, can be done by reasoning within UTP, Hoare Logic, or Dijkstra's Calculus of Weakest Preconditions. Refinement of a TLA specification can be carried out by refinement methods in UTP [117] or using the refinement calculi of Morgan [192] and Back [15].

Note that an atomic action does not have to be implemented by a sequential command. It can be implemented as a piece of concurrent program, say, written in Back's action systems [15].

Exercise 5. Relating different formal notations

1. Specify sequential composition as TLA action?

   $$\tau_1;\ \tau_2$$

   where each action is treated as an atomic action.
   What about when the whole composed $< \tau_1;\ \tau_2 >$ is treated as an atomic action?

2. Model the conditional choice as a TLA action

   if b_1 **then** τ_1 **else** τ_2

3. Define Hoare triple $\{p\}\tau\{q\}$ as a TLA action.
4. Define the Morgan's specification statement $w : [p;\ q]$ as a TLA action.
5. Define the non-deterministic choice $\tau_1 \sqcap \tau_2$ in TLA.
6. Understand how does the TLA notation unify the semantics of deterministic choice and non-deterministic choice.

4 Fault-Tolerance

There are several different ways in which a program can be developed using formal rules which guarantee that it will *satisfy* a specification when executed on an fault-free system [142, 15, 144]. However, when a component of a computer system fails, it will usually produce some undesirable effects and it can be said to no longer behave according to its specification. Such a breakdown of a component is called a fault and its consequence is called a failure. A fault may occur sporadically, or it may be stable and cause the component to fail permanently. Even when it occurs instantaneously, a fault such as a memory fault may have consequences that manifest themselves after a considerable time.

4.1 Introduction

Fault-tolerance is the ability of a system to function correctly despite the occurrence of faults. Faults caused by errors (or "bugs") in software are systematic and can be reproduced in the right conditions. Formal methods can be used to address the problem of errors in software and, while their use does not guarantee the absence of software errors, they do provide the means of making a rigorous, additional check. Hardware errors may also be systematic but in addition they can have random causes. The fact that a hardware component functions correctly at some time is no guarantee of flawless future behaviour. Note that hardware faults often affect the correct behaviour of software.

Of course, it is not possible to tolerate every fault. A failure hypothesis stipulates how faults affect the behaviour of a system. An example of a failure hypothesis is the assumption that a communication medium might corrupt messages. With triple modular redundancy, a single component is replaced by three replicas and a voter that determines the outcome, and the failure hypothesis is that at any time at most one replica is affected by faults. A failure hypothesis divides abnormal behaviour, that is the behaviour that does not conform to the specification, into exceptional and catastrophic behaviours. Exceptional behaviour conforms to the failure hypothesis and must be tolerated, but no attempt need be made to handle catastrophic behaviour (and, indeed, no attempt may be possible). For example, if the communication medium mentioned earlier repeatedly sends the same message, then this may be catastrophic for a given fault-tolerance scheme. It is important to note that "normal" behaviour does

not mean "perfect" behaviour: after a time-out occurs, the retransmission of a message by a sender is normal but it may result in two copies of the same message reaching its destination. Exceptional and normal behaviours together form the acceptable behaviour that the system must tolerate.

Fault-tolerant programs are required for applications where it is essential that *faults* do not cause a program to have unpredictable execution behaviour. We assume that the failures do not arise from design faults in the program, since methods such as those mentioned above can be used to construct error-free programs. So, the only faults we shall consider are those caused by hardware and system malfunctions or the environment of the component that is under development. Many such failures can be *masked* from the program using automatic error detection and correction methods, but there is a limit to the extent to which this can be achieved at reasonable cost in terms of the resources and the time needed for correction.

When the nature or frequency of the errors makes automatic detection *and* correction infeasible, it may still be possible that error *detection* can be performed. It is desirable that fault-tolerant programs are able to perform predictably under these conditions: for example when using memory with single bit error correction and double bit error detection which operates even when the error correction is not effective. In fact, the provision of good program level fault-tolerance can make it possible to reduce the amount of expensive system error correction needed, as program level error recovery can often be focussed more precisely on the damage caused by an error than a general-purpose error correction mechanism.

The task is then to develop programs which perform predictably in the presence of *detected* system errors, and this requires the representation of such errors in the execution of a program. Earlier attempts to use formal proof methods for verifying the properties of fault-tolerant programs were based on an *informal* description of the effects of faults, and this limits their applicability. Here we shall instead model a fault as an *action* which performs state transformations in the same way as other program actions, making it possible to extend a semantic model to include fault actions and to use a single consistent method of reasoning for both program and fault actions.

Let P be a program satisfying the specification Sp. Let the effect of each physical fault in the system on which P is executed be described as a *fault action* which transforms a *good* program state into an *error* state which violates Sp. Physical faults are then modelled as the actions of a *fault program* F which interferes with the execution of P. A *failure* at any point during the execution of P takes it into an error state (F is assumed not to change an error state into a good state.).

In general a high level specification of a program is not sufficient to specify its behaviour in the presence of system faults or to transform it into a fault-tolerant program. It is also necessary to describe the hardware organisation of the system on which the program is to be executed, on its use of the resources of the system and the nature of the possible faults in the system, for instance, which processors

and channels may fail; all of these factors can affect the execution of the program. Very little can be said about the effects of a system fault on a program until it has been refined to the level where these effects can be observed. There is need to represent faults and their effects at various levels of abstraction and here we shall use specifications to develop both the program and the fault environment in which it executes.

4.2 Formal Specification, Verification and Refinement of Fault-Tolerant Programs

A *physical fault* occurring during the execution of a program $P = (\overline{v}, \overline{x}, \Theta, A)$ can cause a transition from a valid state of P into an *error* state. This may lead to a *failure* state which violates the specification of P. A physical fault can be modelled as an atomic *fault-action*.

For example, a malicious fault may set the variables of P to arbitrary values, a crash in a processor may cause variables to become unavailable, and a fault may cause the loss of a message from a channel. Physical faults can thus be described by a set, F, of atomic actions which interfere with the execution of P by possibly changing the values of variables in $\overline{v}$. The *fault-environment* F can be specified by the action formula $\mathcal{N}_F$ which is the disjunction of the action formulas of all $\tau \in F$.

Executing $P = (\overline{v}, \overline{x}, \Theta, A)$ on a system with a fault-environment F is equivalent to interleaving the execution of the actions of P and F. Therefore, interference by F on the execution of P can be defined as a transformation $\mathcal{F}$:

$$\mathcal{F}(P, F) \stackrel{\Delta}{=} (\overline{v}, \overline{x}, \Theta, A \cup F)$$

The exact and canonical specifications of the computations of P when executed on a system with faults F are given by:

$$\Pi(\mathcal{F}(P, F)) = \Theta \wedge \Box[\mathcal{N}_P \vee \mathcal{N}_F]_{\overline{v}}$$
$$\Phi(\mathcal{F}(P, F)) = \exists \overline{x} \cdot \Theta \wedge \Box[\mathcal{N}_P \vee \mathcal{N}_F]_{\overline{v}}$$

Definition 6. *The fault-prone properties of P under F can be derived from the properties of $\mathcal{F}(P, F)$, the F-***affected version** *of P. A computation of $\mathcal{F}(P, F)$ is an F-***affected computation** *of P.*

4.3 Running Example Continued

For the processor-memory interface, assume that the memory is faulty and that its value may be corrupted. Such a fault can be represented by the atomic operation

$$\mathit{fault} \stackrel{\Delta}{=} d' \neq d$$

Let the fault-environment F_1 contain the single action *fault*. The F_1-affected version of P_1 is then:

$$\mathcal{F}(P_1, F_1) = (\Theta_1, \{ R_1^p, W_1^p, R_1^m, W_1^m, \mathit{fault} \})$$

Thus, $\mathcal{N}_{F_1} = \mathit{fault}$ and:

$$\begin{aligned}\mathcal{N}_{\mathcal{F}(P_1,F_1)} &= \mathcal{N}_{P_1} \vee \mathit{fault}\\ \Pi(\mathcal{F}(P_1,F_1)) &= \Theta_1 \wedge \Box[\mathcal{N}_{\mathcal{F}(P_1,F_1)}]_{\overline{v}_1}\\ \Phi(\mathcal{F}(P_1,F_1)) &= \exists\, d \cdot \Pi(\mathcal{F}(P_1,F_1))\end{aligned}$$

For a program P to tolerate a set F of faults, *correcting actions* must be carried out to prevent an *error state* entered by a fault transition from leading to a *failure state* whose occurrence will violate the program requirement specification. In the example, the F_1-affected version $\mathcal{F}(P_1, F_1)$ of P_1 is not a refinement of P_1 and this implies that P_1 does not tolerate the fault F_1.

Definition 7. *For a given set F of faults, a program P is called a* F-tolerant implementation *of a property (or requirement) φ, if $\mathcal{F}(P, F)$ is an implementation of φ:*

$$\Phi(\mathcal{F}(P,F)) \Rightarrow \varphi$$

This means that the behaviours of P comply with the specification φ despite the presence of faults F. When such a property φ is a canonical specification of a program P_h,

$$\Phi(P_h) = \exists\, \overline{y} \cdot \Theta_h \wedge \Box[\mathcal{N}_{P_h}]_{\overline{z}}$$

a program P_l is a *F-tolerant refinement* of P_h, denoted $P_h \sqsubseteq_F P_l$, if P_l is a F-tolerant implementation of $\Phi(P_h)$.

In general, a fault-tolerant program can be obtained from a fault-intolerant program P through transformations by [159, 162]

1. Adding *checkpointing operations*: $\mathcal{C}(P) = P_C$,
2. Adding *recovery operations*: $\mathcal{R}(\mathcal{C}(P)) = P_{FT}$.

The checkpointing transformation C and recovery transformation $\mathcal{R}$ are required so that $P \sqsubseteq P\mathcal{F}(P_{FT}, F)$ or $P \sqsubseteq_F P_{FT}$. There are other ways to construct a P_{FT} such that $P \sqsubseteq_F P_{FT}$.

In [159, 164], checkpointing actions and recovery actions are abstractly defined and can be refined to implement different kinds of fault-tolerant mechanisms.

The F-tolerant refinement relation $\sqsubseteq_F$ is stronger than the ordinary refinement relation: if P_l is a F-tolerant refinement of P_h, then P_l is a refinement of P_h but in general the converse is not true. Further, F-tolerant refinement is generally not reflexive but it is transitive: if $P_h \sqsubseteq_F P_l$ and $P_l \sqsubseteq_F P_{ll}$, then $P_h \sqsubseteq_F P_{ll}$. Fault-tolerant refinements are *fault-monotonic*: if $\mathcal{N}_F \Rightarrow \mathcal{N}_{F_1}$ and $P_h \sqsubseteq_{F_1} P_l$, then $P_h \sqsubseteq_F P_l$. This means that a program which tolerates a set of faults also tolerates any subset of these faults. This is easily achieved in a linear time model, as with TLA. For a discussion of fault-monotonicity with a branching time model, see [126].

Realistic modelling usually requires, in addition to the fault-actions, a *behavioural* fault assumption $\mathcal{B}_F$ about the global properties of F, such as the

maximum number of memories corrupted at a time, and the minimum time between faults. This suggests that the exact specification of the F-affected computations of P should in general be specified as $\Pi(\mathcal{F}(P,F)) \wedge \mathcal{B}_F$, and the F-tolerant refinement of P_h by P_l should be proved under the condition $\mathcal{B}_F$:

$$\Pi(\mathcal{F}(P_l,F)) \wedge \mathcal{B}_F \Rightarrow \widetilde{\Pi(P_h)}$$

which is equivalent to $\mathcal{B}_F \Rightarrow (\Pi(\mathcal{F}(P_l,F)) \Rightarrow \widetilde{\Pi(P_h)})$. This indicates that the proof of F-tolerant refinement of P_h by P_l under $\mathcal{B}_F$ can be established by proving initiality-preservation and step-simulation under the assumption $\mathcal{B}_F$. A behavioural fault assumption prevents certain fault transitions from taking place from some states and is thus in general a safety property of the form $\Box\varphi$. Use the equivalence of $\Box\varphi_1 \wedge \Box\varphi_2$ and $\Box(\varphi_1 \wedge \varphi_2)$, the formula $\Box[\mathcal{N}_{P_l} \vee \mathcal{N}_F]_{\overline{v}} \wedge \mathcal{B}_F$ can be transformed into an equivalent formula $\Box[\mathcal{N}_1]_{\overline{v}}$. In fact, as $\mathcal{B}_F$ should not constrain the actions of $P_l, \mathcal{N}_1$ is obtained from $\mathcal{N}_{P_l}$ and $\mathcal{N}_F$ by enhancing the enabling conditions of the fault actions of F according to $\mathcal{B}_F$. For $\Pi(\mathcal{F}(P_l,F)) \wedge \mathcal{B}_F$, there is F_1 such that $\Pi(\mathcal{F}(P_l,F)) \wedge \mathcal{B}_F$ equals $\Pi(\mathcal{F}(P_l,F_1))$. This implies that the behavioural assumption $\mathcal{B}_F$ can be encoded into the set of fault actions and the two standard steps for proving refinement can be directly applied to the transformed specification $\Pi(\mathcal{F}(P_l,F_1))$. These two methods for proving a fault-tolerant refinement will be demonstrated in the example at the end of this section.

The separation of fault actions and behavioural assumptions simplifies the specification of the F-affected computations of program P_l. Further, coding these assumptions into the fault action makes the proof easier.

4.4 Running Example Continued

Let the fault-free memory of the processor-memory interface P_1 be implemented using three memories, such that at any time at most one suffers from faults.

Let d_i, $i = 1, 2, 3$, be the data in the three memories and let memory i be subject to $fault_i$. The variables f_i with value space $\{0, 1\}$ indicate that d_i has been corrupted when $f_i = 1$. The fault actions can be specified as follows:

$fault_i = (d_i' \neq d_i) \wedge (f_i' = 1)$ corrupts d_i
$F_2 \triangleq \{ fault_1, fault_2, fault_3 \}$
$\mathcal{N}_{F_2} = fault_1 \vee fault_2 \vee fault_3$
$\mathcal{B}_{F_2} \triangleq \Box(f_1 + f_2 + f_3 \leq 1)$ at most one corrupted memory at any time

Define the following auxiliary function:

$$vote(x,y,z) \triangleq \begin{cases} x \text{ if } x = y \text{ or } x = z \\ y \text{ if } x \neq y \text{ and } x \neq z \end{cases}$$

A program P_2 which tolerates the faults F_2 by using the *vote* function tomask the corrupted copy of the memory, and its F_2-affected version are specified as follows:

$$
\begin{array}{rll}
\overline{v}_2 \triangleq & \{\, op, val, d_1, d_2, d_3, f_1, f_2, f_3 \,\} & \\
\Theta_2 \triangleq & (op \in \{rdy, r, w\}) \wedge & \text{initially all} \\
& (d_1 = d_2 = d_3) \wedge (\wedge_{i=1}^{3}(d_i \in \mathbf{Z})) & \text{contain the same value} \\
R_2^p \triangleq & (op = rdy) \wedge (op' = r) & \\
W_2^p \triangleq & (op = rdy) \wedge (op' = w) \wedge (val' \in \mathbf{Z}) & \\
R_2^m \triangleq & (op = r) \wedge (op' = rdy) \wedge & \\
& val' = vote(d_1, d_2, d_3) & \text{return the voted value} \\
W_2^m \triangleq & (\bigwedge_{3}^{i=1}(op = w)) \wedge (op' = rdy) \wedge & \\
& \bigwedge_{i=1}^{3}(d_i' = val) \wedge & \text{write simultaneously} \\
& \bigwedge_{i=1}^{3}(f_i' = 0) & \text{overwrite corrupted copy}
\end{array}
$$

$$
\begin{array}{rll}
A_2 = & \{\, R_2^p, W_2^p, R_2^m, W_2^m \,\} & \text{all actions} \\
P_2 = & (\overline{v}_2, \Theta_2, A_2) & \text{program} \\
\mathcal{N}_{P_2} = & R_2^p \vee W_2^p \vee R_2^m \vee W_2^m & \text{next-state relation} \\
\Pi(P_2) = & \Theta_2 \wedge \Box[\mathcal{N}_{P_2}]_{\overline{v}_2} & \text{exact specification} \\
\Phi(P_2) = & \exists(d_1, d_2, d_3, f_1, f_2, f_3) \cdot \Pi(P_2) & \text{canonical specification}
\end{array}
$$

$$
\begin{array}{rll}
\mathcal{F}(P_2, F_2) = & (\overline{v}_2, \Theta_2, A_2 \cup F_2) & \text{fault-affected program} \\
\mathcal{N}_{\mathcal{F}(P_2,F_2)} = & \mathcal{N}_{P_2} \vee \mathcal{N}_{F_2} & \\
\Pi(\mathcal{F}(P_2, F_2)) = & \Theta_2 \wedge \Box[\mathcal{N}_{P_2} \vee \mathcal{N}_{F_2}]_{\overline{v}_2} & \\
\Phi(\mathcal{F}(P_2, F_2)) = & \exists(d_1, d_2, d_3, f_1, f_2, f_3) \cdot \Pi(\mathcal{F}(P_2, F_2)) &
\end{array}
$$

To prove the refinement relation $P_1 \sqsubseteq_{F_2} P_2$ under the assumption $\mathcal{B}_{F_2}$, define the mapping from the states of $\overline{v}_2$ to those of d: $\widetilde{d} = vote(d_1, d_2, d_3)$. Then, according to the definition of fault-tolerant refinement, we need to prove $\Pi(\mathcal{F}(P_2, F_2)) \wedge \mathcal{B}_{F_2} \Rightarrow \widetilde{\Pi(P_1)}$, where $\widetilde{\Pi(P_1)} = \Pi(P_1)[\tilde{d}/d]$, obtained by substituting $\tilde{d}$ for all occurrences of d in $\Pi(P_1)$.

Proof (of the F_2-tolerant Refinement). The initiality-preservation $\Theta_2 \Rightarrow \widetilde{\Theta}_1$ holds trivially as $\tilde{d} = vote(d_1, d_2, d_3)$, by definition. For step-simulation, we have:

Case 1: R_2^p and W_2^p, and R_2^m equal $\widetilde{R}_1^p$, $\widetilde{W}_1^p$ and $\widetilde{R_2^m}$, respectively;

Case 2: $W_2^m \Rightarrow \widetilde{W}_1^m$, as the right hand side is

$$(op = w) \wedge (op' = rdy) \wedge (vote(d_1', d_2', d_3') = val')$$

Case 3: No $fault_i$-step, for $i = 1, 2, 3$, changes the values val and op, and it is sufficient to show that no $fault_i$-step changes $\tilde{d}$. We prove this for $i = 1$;

the proofs for $i = 2, 3$ are similar. By the assumption $\mathcal{B}_{F_2}$ and the TLA rule for proving an invariance property, it is follows that $\mathcal{F}(P_2, F_2)$ has the following invariance property

$$\Box(f_i = 1 \Rightarrow (d_2 = d_3))$$

Thus, $fault_1 \Rightarrow (d_2' = d_2) \wedge (d_3' = d_3) \wedge (d_2' = d_3')$ and this implies $fault_1 \Rightarrow unchanged(\tilde{d})$.

Exercise 6. Show the fault-tolerant refinement as follows.

- First transform $\Theta_2 \wedge \Box[\mathcal{N}_{P_2} \vee \mathcal{N}_{F_2}]_{\overline{v}_2} \wedge \mathcal{B}_{F_2}$ into

$$\Pi(\mathcal{F}(P_2, F_{21})) \stackrel{\Delta}{=} \Theta_2 \wedge \Box[\mathcal{N}_{P_2} \vee \mathcal{N}_{F_{21}}]_{\overline{v}_2}$$

  where $F_{21} = \{\, fault_{2i} \mid i = 1, 2, 3 \,\}$ and

$$fault_{2i} \stackrel{\Delta}{=} (f_{i\oplus 1} = 0 \wedge f_{i\oplus 2} = 0) \wedge (d_i' \neq d_i) \wedge (f_i' = 1)$$

  where $\oplus$ is + modulo 3.
- Then prove $\Pi(\mathcal{F}(P_2, F_{21})) \Rightarrow \widetilde{\Pi(P_1)}$ by establishing initiality-preservation and step-simulation.

5 Modelling Real-Time Programs

The most common timing constraints over a program require its actions to be executed neither *too early* nor *too late*; for example, to use time for the synchronization between a processor and a memory to ensure that a message written is not overwritten before being read, the memory must not execute the *read* operation too slowly and the processor must not issue the *write* operation too soon. Let time be represented by the non-negative real numbers $\mathbf{R}^+$. However, it is not difficult to see that the methods and results apply to discrete time domains as well.

We now specify timing constraints over the execution of an action in a program P can be specified by assigning to each action τ a volatile lower time bound $L(\tau)$ from $\mathbf{R}^+$ and a volatile upper time bound $U(\tau)$ which is either a value from $\mathbf{R}^+$, or the special value ∞ which denotes the absence of an upper bound. Any real number in $\mathbf{R}^+$ is assumed to be less than ∞, and the lower bound is assumed not to exceed the upper bound for any action.

Definition 8. *A* **real-time program** *can be represented as* $P^T = \langle P, L, U \rangle$, *where* P *is an* untimed *program, defined in the previous section, and* L *and* U *are functions of the atomic actions of* P *defining the lower bound* $L(\tau)$ *and upper bound* $U(\tau)$ *for any action* τ *of* P.

5.1 Specifying Real Time

As in the case of untimed programs, we shall need an exact specification $\Pi(P^T)$ of a real-time program P^T. We introduce a distinguished state variable *now* to represent time that is is also known as the global clock, and an action to advance time, under the following assumptions [2, 110]:

time starts at 0: initially $now = 0$.
time never decreases: $\Box[now' \in (now, \infty)]_{now}$.
time diverges: $\forall\, t \in \mathbf{R}^+ \cdot \Diamond(now > t)$.

Time divergence is also called the Non-Zeno property and ensures that only a finite number of actions can be performed in any finite interval of time. The three assumptions can be combined to specify real-time evolution:

$$RT \triangleq (now = 0) \wedge \Box[now' \in (now, \infty)]_{now} \wedge \forall\, t \in \mathbf{R}^+ \cdot \Diamond(now > t)$$

To preserve the atomicity of the actions in the program, we model the execution of the program so that program state and time do not change simultaneously and that a program state can be changed only by program actions. This is specified by the condition $\tau \Rightarrow (now' = now)$ for each action τ of P. Then the conjunction $\Pi(P) \wedge RT$ specifies the interleaving of program actions and time evolution. The program actions are further constrained by their *lower bound* and *upper bound* conditions, and this is done by introducing auxiliary state variables called *timers*.

5.2 Specifying Time Bounds

An action τ in P^T cannot take place before it has been enabled for $L(\tau)$ time units and must take place before it has been enabled for more than $U(\tau)$ units. We need to introduce auxiliary state variables to record how long an action has been enabled.

Consider the hour-clock again. Assume it is now required that the clock display the *correct real time*. Following what has been described earlier, we have:

- The newly added observable variable, *now*, representing time.
- The change of the display is instantaneous. For example

$$\left\{\begin{matrix} hr \mapsto 12, \\ now \mapsto \sqrt{2.47} \end{matrix}\right\}, \left\{\begin{matrix} hr \mapsto 12, \\ now \mapsto \sqrt{2.5} \end{matrix}\right\}, \cdots,$$

- *now* changes between of a change of display.
- The requirement that the interval between two ticks is one hour plus or minus ρ seconds.
- The need of a timer t to record how much time has elapsed since the last tick.

$$tNxt(HCnxt) \triangleq (t' = 0) \lhd HCnxt \rhd (t' = t + (now' - now))$$
$$Timer(t, HCnxt) \triangleq (t = 0) \wedge \Box[tNxt]_{(t, hr, now)}$$

– The timer t cannot exceed $360 + \rho$ before the next tick:

$$Max(t, 360 + \rho) \stackrel{\Delta}{=} \Box(t \leq 360 + \rho)$$

– After a tick, the clock cannot tick again before t becomes $360 + \rho$:

$$Min(t, HCnxt, 360 + \rho) \stackrel{\Delta}{=} \Box[HCnxt \Rightarrow (t \geq 360 - \rho)]_{hr}$$

– The time bound specification HC_B is then the conjunction:

$$HC_B \stackrel{\Delta}{=} Timer(t, HCnxt) \wedge Maxt, 360 + \rho) \wedge Min(t, HCnxt, 360 + \rho)$$

The exact specification of the real-time clock is:

$$RTHC \stackrel{\Delta}{=} HC \wedge RT \wedge HC_B$$

Definition 9. *In general, given a program $P = (\overline{v}, \overline{x}, \Theta, A)$, let $\tau \in A$ and δ be a non-negative real. We can define a* **counting-up volatile timer***:*

$$\begin{aligned} Timer(t_\tau, \tau) \stackrel{\Delta}{=} & (t = 0) \wedge \Box[(t' = 0) \lhd (< \tau >_{\overline{v}} \vee \neg en(\tau)') \rhd \\ & (t'_\tau = t_\tau + (now' - now))]_{(t_\tau, \overline{v}, now)} \end{aligned}$$

where $A_1 \lhd g \rhd A_2$ denotes the action $g \wedge A_1 \vee \neg g \wedge \neg A_2$.

Then we can specify the time bounds of a real-time version P^T of program P as follows:

$$\begin{aligned} Max \uparrow (\tau) &\stackrel{\Delta}{=} \Box(t_\tau \leq U(\tau)) \\ Min \uparrow (\tau) &\stackrel{\Delta}{=} \Box[\tau \Rightarrow t \geq L(\tau)]_{(\overline{v}, now)} \\ B_\tau \uparrow \quad &\stackrel{\Delta}{=} Timer(t_\tau, \tau) \wedge Min \uparrow (\tau) \wedge Max \uparrow (\tau) \\ B_P \uparrow \quad &\stackrel{\Delta}{=} \bigwedge_{\tau \in A} B_\tau \end{aligned}$$

The exact specification of P^T can be given as $\Pi(P) \wedge RT \wedge B_P \uparrow$.

Using counting-up timers gives a simpler specification of a real-time program. However, our experience is that with them, the proof of a refinement is hard. We now define a *counter-down timer.*

Definition 10. *Given a program $P = (\overline{v}, \overline{x}, \Theta, A)$, let $\tau \in A$ and δ be a non-negative real. We define* **volatile δ-timer** *t which is a state variable not in $\overline{v}$. The behaviour of the timer t is such that when τ is enabled from a state in which it was disabled or τ is taken, t is assigned a* clock time *of now$+\delta$ units of time:*

$$\begin{aligned} Volatile(t, \tau, \delta, \overline{v}) \stackrel{\Delta}{=} & ((t = \delta) \lhd en(\tau) \rhd (t = \infty)) \wedge \\ & \Box[\quad (en(\tau)' \wedge (\tau \vee \neg en(\tau)) \wedge (t' = now + \delta) \\ & \quad \vee en(\tau) \wedge en(\tau)' \wedge \neg\tau \wedge (t' = t) \\ & \quad \vee \neg en(\tau)' \wedge (t' = \infty)) \wedge (\overline{v}, now)' \neq (\overline{v}, now)]_{(t, \overline{v})} \end{aligned}$$

Informally, each line in the definition is explained as: the volatile δ-timer t is initially set to δ (that is, δ time units ahead of the initial value *0* of *now*) if τ is enabled, and to ∞ otherwise, and then repeated in every step:

1. the timer t is reset to δ time units ahead of *now* in the new state if:
   (a) τ becomes enabled in the new state from being disabled in the old state, or
   (b) τ is taken and it remains enabled in the new state;
2. the timer t stays unchanged if τ remains enabled but τ has not taken place in this step;
3. the timer t is reset to ∞ if τ is disabled in the new state.

Using such a volatile timer t, the property that a τ-step cannot take place until the time *now* reaches the clock time t can be defined as:

$$\mathit{MinTime}(t, \tau, \overline{v}) \triangleq \Box[\tau \Rightarrow (t \leq \mathit{now})]_{\overline{v}}$$

The conjunction of this formula and $\mathit{Volatile}(t, \tau, \delta, \overline{v})$ can be used to specify a lower bound condition; and $\mathit{Volatile}(t, \tau, \delta, \overline{v})$ can be used also for an upper bound when conjoined with the formula:

$$\mathit{MaxTime}(t) \triangleq \Box[\mathit{now}' \leq t]_{\mathit{now}}$$

For a given real-time program $P^T = \langle P, L, U \rangle$, let each action τ of P have a volatile $L(\tau)$-timer t_τ and volatile $U(\tau)$-timer T_τ. Then the conjunction:

$$\mathit{Volatile}(t_\tau, \tau, L(\tau), \overline{v}) \wedge \mathit{MinTime}(t_\tau, \tau, \overline{v})$$

which is *true* when $L(\tau) = 0$, specifies the lower bound for action τ. A τ-step cannot take place within $L(\tau)$ time units of when τ becomes enabled, and the next τ step cannot occur within $L(\tau)$ time units of when τ is re-enabled. The lower bound condition of the program is the conjunction of the lower bound conditions for all its actions:

$$\mathit{LB}(P^T) \triangleq \bigwedge_{\tau \in A} (\mathit{Volatile}(t_\tau, \tau, L(\tau), \overline{v}) \wedge \mathit{MinTime}(t_\tau, \tau, \overline{v}))$$

Similarly, the upper bound condition of program P^T is specified by the formula:

$$\mathit{UB}(P^T) \triangleq \bigwedge_{\tau \in A} (\mathit{Volatile}(T_\tau, \tau, U(\tau), \overline{v}) \wedge \mathit{MaxTime}(T_\tau))$$

where $\mathit{Volatile}(T_\tau, \tau, U(\tau), \overline{v}) \wedge \mathit{MaxTime}(T_\tau)$ equals *true* and thus can be eliminated from the conjunction if $U(\tau) = \infty$.

The time bound specification $B(P^T)$ for the whole program P^T is then the conjunction $\mathit{LB}(P^T) \wedge \mathit{UB}(P^T)$.

Definition 11. *The real-time executions of program P^T are exactly specified by the* **exact specification***:*

$$\Pi(P^T) \triangleq \Pi(P) \wedge RT \wedge B(P^T)$$

Hiding the internal variables $\overline{x}$ *and the set of auxiliary timers, which is denoted by* $timer(P^T)$*, gives the* **canonical specification** *of* P^T*:*

$$\Phi(P^T) \stackrel{\Delta}{=} \exists\, \overline{x}, timer(P^T) \cdot \Pi(P^T)$$

5.3 Running Example Continued

In the untimed processor-memory interface P_1, let the processor and the memory be synchronized by timing rather than by guarding the processor actions. Assume that the processor periodically issues an operation every ρ units of time. To ensure that an operation is executed by the memory before the next operation is issued by the processor, ρ must be greater than the upper bound (or deadline) for the memory to execute the operation. The real-time program $P_1^T = \langle P_1, L_1, U_1 \rangle$ is described as follows:

$$\begin{aligned}
\overline{v}_1 &\stackrel{\Delta}{=} \{op, val, d, c\} && \text{add an internal variable } c \\
\Theta_1 &\stackrel{\Delta}{=} (op \in \{r, w\}) \wedge (d \in \mathbf{Z}) \wedge \neg c && \text{the op has not been completed} \\
RW_1^p &\stackrel{\Delta}{=} (op' = r) \wedge \neg c' \vee && \text{issues a read operation, or} \\
&\quad\ (op' = w) \wedge \neg c' \wedge (val' \in \mathbf{Z}) && \text{a write operation} \\
R_1^m &\stackrel{\Delta}{=} (op = r) \wedge \neg c \wedge (val' = d) \wedge c' && \text{similar to the original } P_1 \\
W_1^m &\stackrel{\Delta}{=} (op = w) \wedge \neg c \wedge (d' = val) \wedge c' && \text{similar to the original } P_1
\end{aligned}$$

$$\begin{aligned}
A_1 &\stackrel{\Delta}{=} \{RW_1^p, R_1^m, W_1^m\} && \text{actions of the program} \\
L_1(RW_1^p) &= U_1(RW_1^p) = \rho && RW_1^p\text{'s period} \\
L_1(R_1^m) &= L_1(W_1^m) = 0 && \text{memory actions' lower bound} \\
U_1(R_1^m) &= U_1(W_1^m) = D_1 < \rho && \text{memory actions' upper bound} \\
P_1^T &= \langle \overline{v}_1, \Theta_1, A_1, L_1, U_1 \rangle && \text{real-time program}
\end{aligned}$$

5.4 Verification and Refinement

The timed and untimed properties of programs can be specified in the same way in TLA. For example, the bounded response property that once φ occurs in an execution, ψ must occur within δ time units can be described as:

$$\varphi \stackrel{\delta}{\rightsquigarrow} \psi \stackrel{\Delta}{=} \forall\, t \cdot \Box(\varphi \wedge now = t \Rightarrow \Diamond(\psi \wedge now \leq t + \delta))$$

To prove that the real-time program P^T *satisfies* (or *implements*) a timing property is to prove the implication of the property by the specification $\Phi(P)$ of the program. For example, the real-time processor-memory interface P_1^T satisfies the property:

$$\exists\, d \cdot ((op = r \wedge d = v) \stackrel{D_1}{\rightsquigarrow} (val = v))$$

which asserts that the value of the memory will be output within D_1 units of time after the processor issues a read operation. The implication:

$$\Phi(P_1^T) \Rightarrow \exists\, d \cdot ((op = r \wedge d = v) \stackrel{D_1}{\rightsquigarrow} (val = v))$$

can be proved by proving:

$$\Pi(P_1^T) \Rightarrow (op = r \wedge d = v) \overset{D_1}{\rightsquigarrow} (val = v)$$

Definition 12. *The* **refinement relation** $P_h^T \sqsubseteq P_l^T$ *between the real-time programs* P_l^T *and* P_h^T *is defined as the implication* $\Phi(P_l^T) \Rightarrow \Phi(P_h^T)$ *using a refinement mapping.*

To verify initiality-preservation and step-simulation, convert the exact specification:

$$\Pi(P^T) \triangleq \Theta_P \wedge [\mathcal{N}]_{\overline{v}} \wedge RT \wedge B(P^T)$$

into the form $\Theta \wedge [\mathcal{N}]_{\overline{z}}$, where $\overline{z}$ equals $\overline{v}$ plus *now* and the timers, and Θ is obtained from Θ_P by conjoining it with the initial conditions on *now* and the timers. $\mathcal{N}$ is an action formula.

Exercise 7. Consider the Gas Burner example in [223, 143]. This case study formulates the safety requirement of a gas burner in terms of a variable *Leak* denoting an undesirable but unavoidable state which represents the presence of unlit gas.

For safety, *gas must never leak for more than 4 seconds in any period of at most 30 seconds*. This is specified by the bounded critical duration property. To meet the requirement *Req*, two design decisions are made:

Des-1 any occurrence of leak must be stopped within 4 seconds, and
Des-2 two occurrences of leaks must be separated by a period of 26 seconds in which the burner does not leak; in other words, a *Leak* is stopped it may not reoccur within 26 seconds.

1. Write TLA specifications for the above design decisions.
2. Reason within TLA that the above two design decisions are met by the timed transition system defined below:

$$\begin{aligned} GB_1 = \langle & \Theta_1 : true \\ & \tau_1 : Leak \wedge \neg Leak'[0, 4] \\ & \tau_2 : \neg Leak \wedge Leak'[26, \infty) \\ & \rangle \end{aligned}$$

3. After the initial design, GB_1 can be refined. For example, the transition system GB_2 in Figure 3 is a refinement of GB_1. GB_2 has the following *phases*:
   Idle: Await heat request with no gas and no ignition. It enters **Purge** within e time units on heat request. The parameter e in this example is the system wide upper bound for *reactions*.
   Purge: Pauses for 30 seconds and then enters **Ignite1** within e time units.
   Ignite1: Turns on ignition and gas and after one second exits within e to **Ignite2**.

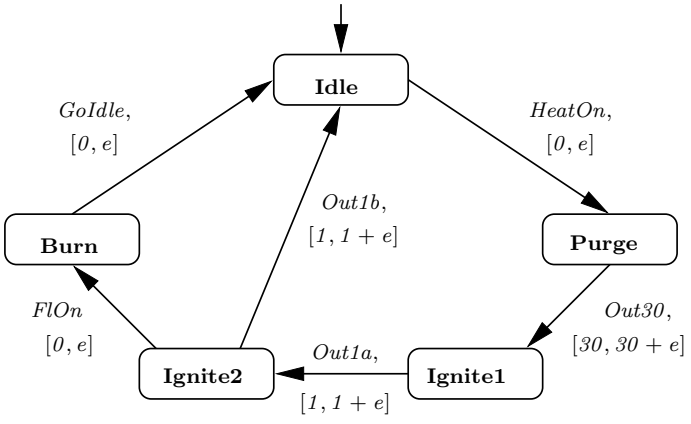


Fig. 3. A refinement of GB_1

Ignite2: Monitors the flame, if it is sensed within one second **Burn** is entered, otherwise it returns to **Idle** within e while turning the gas off.

Burn: Ignition is switched off, but gas is still on. The **Burn** phase is stable until heat request goes off. Gas is then turned off and **Idle** is entered within e.

This refinement uses a simple error recovery: return to **Idle** from **Ignite2**. We assume no flame failure in the **Burn** phase. Therefore, in this implementation, *Leak* can only occur in the **Ignite1** and **Ignite2** phases.

4. Formalize in TLA the full and canonical specification of GB_2, and decide the constant e such that GB_2 also meets the two design decisions for GB_1.

6 Combining Fault-Tolerance and Timing Properties

Fault-tolerant systems often have real-time constraints. So it is important that the timing properties of a program are refined along with the functional and fault-tolerant properties defined in the program specification. This section extends the transformational approach for fault-tolerance by adding time bounds to actions. This will allow the fault-tolerant redundant actions to be specified with time constraints.

The functional properties of faults are modelled by a set F of atomic actions specified by the action formula $\mathcal{N}_F$. There are no time bounds on these actions (or, equivalently, the lower and upper bound of each fault action are respectively 0 and ∞). Given a real-time program $P^T = \langle P, L, U \rangle$, the F-affected version of P^T is defined as:

$$\mathcal{F}(P^T, F) \triangleq \langle \mathcal{F}(P, F), L, U \rangle$$

where the domain of L and U is extended to $A \cup F$ and each action in F is assigned time bounds of 0 and ∞.

To achieve fault-tolerance in a real-time system, there must be a timing assumption on the occurrence of faults, especially when deadlines are required to be met. Such an assumption is usually a constraint on the frequency of occurrence of faults, or the minimum time for which faults cannot occur. This period should be long enough for the recovery of the computation to take place and for progress to be made after recovery from a fault. For a formula ψ and a non-negative real number ε, let ψ hold continuously for ε units of time:

$$\Box_\varepsilon \psi \stackrel{\Delta}{=} \forall\, t \cdot (now = t \Rightarrow \Box(now \leq t + \varepsilon \Rightarrow \psi))$$

A fault is modelled as an atomic action and specified as an action formula. The timing assumption on faults F is a conjunction of assumptions, each of the form *whenever fault$_1$ occurs, fault$_2$ cannot occur within ε units of time*. If this assumption is denoted by $\mathcal{T}_F$, the exact and canonical specifications of the F-affected version $\mathcal{F}(P^T, F)$ are, respectively,

$$\begin{aligned} \Pi(\mathcal{F}(P^T, F)) &= \Theta \wedge [(\mathcal{N}_P \vee \mathcal{N}_F)]_{\overline{v}} \wedge RT \wedge B(P^T) \wedge \mathcal{B}_F \wedge \mathcal{T}_F \\ &= \Pi(\mathcal{F}(P, F)) \wedge RT \wedge B(P^T) \wedge \mathcal{B}_F \wedge \mathcal{T}_F \\ \Phi(\mathcal{F}(P^T, F)) &= \exists\, \overline{x}, timer(P^T) \cdot \Pi(\mathcal{F}(P^T, F)) \end{aligned}$$

Thus the F-affected version of a real-time program P^T is also a real-time program. This normal form allows the definition of *fault-tolerance for real-time systems* to be given in the same way as for untimed systems. A real-time program P^T is an *F-tolerant implementation* of a real-time property ψ if the implication $\Phi(\mathcal{F}(P^T, F)) \Rightarrow \psi$ holds. P^T is an *F-tolerant refinement* of a real-time program P_h^T if the implication $\Phi(\mathcal{F}(P^T, F)) \Rightarrow \Phi(P_h^T)$ holds.

6.1 Running Example Continued

In Section 4, we showed how the untimed fault-free processor-memory interface P_1 can be implemented by the untimed version of P_2, using three faulty memories whose values may be corrupted by the set F_2 of faults with assumption $\mathcal{B}_{F_2}$. We show now how the timed version P_1^T is F_2-tolerantly refined by a timed version of P_2.

Let the specification of the underlying untimed program P_2 be changed slightly by removing the guard condition of the processor actions:

$$\begin{aligned} \overline{v}_2 &= \{\, op, val, c, d_1, d_2, d_3, f_1, f_2, f_3 \,\} \\ \Theta_2 &= (op \in \{r, w\}) \wedge \neg c \\ &\quad (d_1 = d_2 = d_3) \wedge (\bigwedge_{i=1}^{3} (d_i \in \mathbf{Z})) \\ RW_2^p &= (op' = r) \wedge \neg c' \vee (op' = w) \wedge \neg c' \wedge (val' \in \mathbf{Z}) \\ R_2^m &= (op = r) \wedge \neg c \wedge c' \wedge val' = vote(d_1, d_2, d_3) \\ W_2^m &= (op = w) \wedge \neg c \wedge c' \wedge \bigwedge_{i=1}^{3} (d_i' = val \wedge f_i' = 0) \\ \mathcal{N}_{P_2} &= RW_2^p \vee R_2^m \vee W_2^m \end{aligned}$$

$$\Pi(P_2) = \Theta_2 \wedge \Box[\mathcal{N}_{P_2}]_{\overline{v}_2}$$
$$\Phi(P_2) = \exists(d_1, d_2, d_3, c, f_1, f_2, f_3) \cdot \Pi(P_2)$$

Meeting the timing properties of P_1^T requires that the time bounds of the actions of the implementation P_2 guarantee (a) that the period for the processor to issue an operation is still ρ and (b) that the upper bound D_2 for the memory to execute an operation to completion is not greater than D_1:

$$L_2(RW_2^p) = U_2(RW_2^p) = \rho$$
$$L_2(R_2^m) = L_2(W_2^p) = 0$$
$$U_2(R_2^m) = U_2(W_2^m) = D_2 \leq D_1$$

To prove that $P_1^T \sqsubseteq_{F_2} P_2^T$ under the assumption $\mathcal{B}_{F_2}$, we have to consider only the case when $D_2 = D_1$ since simply lowering the upper bound (or raising the lower bound) of an action is obviously a refinement. Define a refinement mapping from the states over the variables of P_2^T's to the states over the internal variables of P_1^T, including the volatile timers as follows:

$$\tilde{d} \triangleq vote(d_1, d_2, d_3) \qquad \tilde{c} \triangleq c$$
$$\widetilde{t_{RW_1^p}} \triangleq t_{RW_2^p} \qquad \widetilde{T_{RW_1^p}} \triangleq T_{RW_2^p}$$
$$\widetilde{T_{R_1^m}} \triangleq t_{R_2^m} \qquad \widetilde{T_{W_1^m}} \triangleq T_{W_2^m}$$

The implication $\Pi(P_2^T, F_2) \wedge \mathcal{B}_{F_2} \Rightarrow \widetilde{\Pi(P_1^T)}$ can be proved in the same way as for the untimed fault-tolerance in Section 4.

The assumption $\mathcal{B}_{F_2}$ can be relaxed to the timing assumption:

$$\mathcal{T}_{F_2} \triangleq \bigwedge_{i=1}^{3} \Box(f_i = 1 \Rightarrow \Box_{\rho + D_2}(f_{i \oplus 1} = 0 \wedge f_{i \oplus 2} = 0))$$

which asserts that only one of the most recently written memories may be corrupted before the read operation is completed. Then $P_2^T = \langle P_2, L_2, U_2 \rangle$ is also an F_2-tolerant refinement of P_1^T under the fault-assumption $\mathcal{T}_{F_2}$.

The specifications of P_1^T and P_2^T demonstrate a practical fact: to achieve fault-tolerance with timing constraints, a more powerful (or faster) machine is often needed. The execution of the multiple assignment W_2^m on such a machine should not be slower than the execution of the single assignment W_1^m on a machine for an non-fault-tolerant implementation of P_1^T; and the execution of the multiple read operation R_2^m with a voting function should not be slower than the execution of the single read operation R_1^m. Otherwise, with a machine of the same speed, the original time bounds must have enough slack to accommodate the redundant actions for fault-tolerance.

We can refine P_2^T further to P_3^T, where the actions of the three memories are executed by different processes, and the voting action is done by another process. The specification of the variables, the initial condition and the actions of P_3 are given below, for $i = 1, 2, 3$:

$$
\begin{aligned}
\overline{v}_3 &= \{\, op, val, op_i, val_i, d_i, f_i, c_i, v_i \mid i = 1, 2, 3 \,\} \\
\Theta_3 &= (op \in \{r, w\}) \wedge \neg c_1 \wedge \neg c_2 \wedge \neg c_3 \wedge \\
&\quad (op_1 = op_2 = op_3 = op) \wedge \neg v_1 \wedge \neg v_2 \wedge \neg v_3 \\
RW_3^p &= \neg c_1' \wedge \neg c_2' \wedge \neg c_3' \wedge ((op, op_1, op_2, op_3)' = (r, r, r, r) \vee \\
&\quad ((op, op_1, op_2, op_3)' = (w, w, w, w)) \wedge (val' \in \mathbf{Z})) \\
R_3^{m_i} &= (op_i = r) \wedge \neg c_i \wedge (val_i' = d_i) \wedge c_i' \wedge v_i \\
W_3^{m_i} &= (op_i = w) \wedge \neg c_i \wedge (d_i' = val) \wedge (f_i' = 0) \wedge c_i' \\
Vote &= v_1 \wedge v_2 \wedge v_3 \wedge (val' = vote(val_1, val_2, val_3) \wedge \\
&\quad \neg v_1' \wedge \neg v_2' \wedge \neg v_3' \\
A_3 &= \{\, RW_3^p, Vote, R_3^{m_i}, W_3^{m_i} \mid i = 1, 2, 3 \,\} \\
P_3 &= (\overline{v_3}, \Theta_3, A_3)
\end{aligned}
$$

The newly introduced internal variables, op_i, contain the operations issued to the process i; c_i denote whether the operations issued to the process i are completed; v_i are used to synchronize the read actions and the vote action such that *vote* is done only after all the *read*s are completed.

The timing properties of P_2^T require (a) that the time bounds of the actions in the implementation P_3^T guarantee that the period for the processor to issue an operation is still ρ, (b) that the upper bound D_{wi} for the ith memory to execute an issued write is not greater than D_2, and (c) that the sum of the upper bound D_{ri} of the ith memory to execute an issued read operation and the upper bound D_{vote} of the *Vote* action is not greater than D_2: for $i = 1, 2, 3$.

$$
\begin{array}{ll}
L_3(RW_3^p) = U_3(RW_3^p) = \rho & \\
L_3(R_3^{m_i}) = L_3(W_3^{m_i}) = L_3(Vote) = 0 & U_3(W_3^{m_i}) = D_{w_i} \\
U_3(R_2^{m_i}) = D_{r_i} & U_3(Vote) = D_{vote} \\
D_{w_i} \leq D_2 & D_{r_i} + D_{vote} \leq D_2
\end{array}
$$

The refinement and fault-tolerance can be proved by showing the validity of the implication:

$$\Phi(\mathcal{F}(P_3^T, F_2)) \wedge \mathcal{B}_{F_2} \Rightarrow \Phi(\mathcal{F}(P_2^T, F_2)) \wedge \mathcal{B}_{F_2}$$

from the following refinement mapping when $D_{w_i} = D_2$ and $D_{r_i} + D_{vote} = D_2$:

$$
\begin{aligned}
\tilde{c} &\triangleq c_1 \wedge c_2 \wedge c_3 \\
\tilde{d}_i &\triangleq \begin{cases} d_i \text{ if } c_1 \wedge c_2 \wedge c_3 \vee \neg c_1 \wedge \neg c_2 \wedge \neg c_3 \\ val \text{otherwise} \end{cases} \\
\widetilde{T_{R_2^{m_i}}} &\triangleq \begin{cases} min\{\, T_{R_3^{m_i}} \mid i = 1, 2, 3 \,\} + D_{vote} & \text{if } \bigvee_{i=1}^{3} (op_i = r \wedge \neg v_i) \\ T_{Vote} & \text{if } v_1 \wedge v_2 \wedge v_3 \\ \infty & \text{otherwise} \end{cases} \\
\widetilde{T_{W_2^{m_i}}} &\triangleq min\{\, T_{W_3^{m_i}} \mid i = 1, 2, 3 \,\}
\end{aligned}
$$

However, it is important to notice that it is easier to understand and prove the F_2-refinement of P_2^T by P_3^T if this refinement is done stepwise:

1. first refine P_2^T into a program P_{31}^T by replacing W_2^m in P_2 with the three write operations $W_3^{m_i}$ and setting $U(W^{m_i}) = D_2,\ i = 1, 2, 3$;
2. then refine P_{31}^T into another program P_{32}^T by replacing R_2^m in P_{31} (which is also in P_2) with the three read operations $R_3^{m_i}$ plus *Vote* and setting $U(R^{m_i}) + U(Vote) = D_2,\ i = 1, 2, 3$;
3. finally, scale down the upper bounds of the new operations to get P_3^T.

7 Feasible Real-Time Scheduling as Refinement

To model the parallel execution of a program P^T, we partition the actions A of P into n sets (*processes*) $p_1, \ldots, p_n$. A *shared state variable* is one which is used by actions in different processes, while a *private state variable* is used only by the actions in one process. Two actions of P can be executed in parallel *iff* they are not in the same process and do not share variables (shared variables are accessed under mutual exclusion). In such a concurrent system, processes communicate by executing actions which use shared variables. We assume that each process in a concurrent program is *sequential*. In other words, at most one atomic action in a process is enabled at a time, though an action of the process may be non-deterministic as it may carry out transitions from the same state to different states in different executions.

Let the real-time program P^T be implemented on a system by assigning its n processes to a set $\{1, \ldots, m\}$ of processors and executing them under a *scheduler*. Such an implementation is *correct* iff it meets both the functional requirements defined by the actions of P and the timing constraints defined by the time bound functions L and U of P^T. Rather than adding scheduling primitives to the programming (specification) language (as in [123]), the program and the scheduler will be modelled and specified in a single semantic model but their correctness will be proved separately. The application of a scheduler to a program on a given set of processors can be described as a transformation of the program, and the schedulability of the program can be determined by reasoning about the transformed or *scheduled* program.

Using transformations and separating the program from the scheduler helps to preserve the independence of the program from scheduling decisions. The programmer does not need to take account of the system and the scheduler until the program is ready to be implemented. This allows the feasibility of a program under different schedulers and the effect of a scheduler on different programs to be investigated. Also, the feasibility of the implementation of a program can be proved by considering a scheduling policy, rather than low-level implementation details.

We shall first describe the functional and timing aspects of a scheduler, and then determine how they affect the execution of the program.

7.1 Untimed Scheduling

Assume that a scheduler allocates a process of P for execution by a processor using a *submit action*, and removes a process from a processor by a *retrieve*

action. We shall say that a process is on a processor if the process has been allocated to that processor.

An atomic action of a process can be executed only when the process is on a processor and the action is enabled. Let the Boolean variable run_i, $1 \leq i \leq n$, be *true* if process p_i is on a processor. The effect of scheduling is represented by a transformation $\mathcal{G}(P)$ in which each atomic action τ of P in the process p_i, $1 \leq i \leq n$, is transformed by strengthening its enabling condition by the Boolean variable run_i. Let $r(\tau)$ denote the transformed action of τ in $\mathcal{G}(P)$. Then:

$$r(\tau) \stackrel{\Delta}{=} run_i \wedge \tau$$

Therefore, $en(r(\tau)) \Leftrightarrow run_i \wedge en(\tau)$, and a process p_i is *being executed* only when it is on a processor and one of its actions is enabled.

A scheduler can be functionally described as an untimed program S whose initial condition $idle \stackrel{\Delta}{=} \forall\, i \cdot \neg run_i$ guarantees that there is no process on any processor and whose submit and retrieve actions modify the variables run_i. We use a *generic* description so that the scheduler can be applied to any program P on a system with any number of processors. Program P and the set of processors will be left as parameters, to be replaced respectively by a concrete program and the definition of a specific system.

Given S, the scheduling of P by S on a set of processors can be described as a transformation $\mathcal{I}(P)$. The initial condition of the scheduled program $\mathcal{I}(P)$ is the conjunction of the initial conditions of S and P. Formally, this is simply described by $idle \wedge \Theta$. The actions of $\mathcal{I}(P)$ are formed by the union of the actions of S and $\mathcal{G}(P)$ and their execution is interleaved.

An execution of $\mathcal{I}(P)$ is a state sequence σ over the union of the state variables $\overline{z}$ of the scheduler and the variables $\overline{v}$ of the program P for which:

1. the initial state σ_0 satisfies the initial conditions Θ of P and *idle* of S,
2. for each step (σ_j, σ_{j+1}), one of the following conditions holds:
   (a) $\sigma_{j+1} = \sigma_j$, or
   (b) σ_{j+1} is produced from σ_j by an action in S, or
   (c) σ_{j+1} is produced by the execution of an action τ in a process p_i whose enabling condition and the predicate run_i are both true in σ_j.

The set of executions of $\mathcal{I}(P)$ is then specified by:

$$\Pi(\mathcal{I}(P)) = idle \wedge \Theta \wedge \Box[\mathcal{N}_{\mathcal{G}(P)} \vee \mathcal{N}_S]_{(\overline{v},\overline{z})}$$

We assume that S does not change the state of P, that is, $\mathcal{N}_S \Rightarrow (\overline{v}' = \overline{v})$. This gives us the compositional specification:

$$\Pi(\mathcal{I}(P)) = \Pi(S) \wedge \Pi(\mathcal{G}(P))$$

Note that $r(\tau) \Rightarrow \tau$ holds for each action τ of P. So does $\Pi(\mathcal{G}(P)) \Rightarrow \Pi(P)$. Hence, $\Pi(\mathcal{I}(P))$ implies $\Pi(P)$. This shows that $\mathcal{I}(P)$ refines P and the transformation $\mathcal{I}$ (and thus the scheduler S) preserves the functional properties of P.

7.2 Timed Scheduling

The timing properties of the executions of $\mathcal{I}(P)$ depend on the number of processors and their execution speed. Assume that the *hard execution time* needed for each atomic operation τ on a processor lies in a real interval $[l(\tau), u(\tau)]$. If the execution of τ on a processor starts at time t and finishes at time $t + d$, then the *total execution time* for τ in the interval $[t, t + d]$ lies in the interval $[l(\tau), u(\tau)]$. The functions l and u define the (persistent) time bounds of the actions in $\mathcal{G}(P)$. The real-time program $\mathcal{G}(P)^T \stackrel{\Delta}{=} \langle \mathcal{G}(P), l, u \rangle$, where for each $r(\tau)$ of $\mathcal{G}(P)$, $l(r(\tau)) = l(\tau)$ and $u(r(\tau)) = u(\tau)$.

To guarantee that the implementation of P^T satisfies its real-time deadlines, the computational overhead of the *submit* and *retrieve* actions must be bounded. Let the scheduler S have time bounds $L_S(\tau)$ and $U_S(\tau)$ for each action τ of S and let the real-time scheduler be S^T.

Definition 13. *The* **real-time scheduled program**

$$\mathcal{I}(P^T) \stackrel{\Delta}{=} \langle \mathcal{I}(P), L_{\mathcal{I}(P)}, U_{\mathcal{I}(P)} \rangle$$

where the functions $L_{\mathcal{I}(P)}$ and $U_{\mathcal{I}(P)}$ are respectively the union of the functions L_S and l, and the union of U_S and u.

Here, for functions p from Set_1 to Set and q from Set_2 to Set, where Set_1 and Set_2 are disjoint, the *union* of p and q is the function from $Set_1 \cup Set_2$ to Set that equals p for elements in Set_1 and q for elements in Set_2.

The definition states that the execution speed of the processors and the timing properties of the scheduler determine the timing properties of the scheduled program.

As the actions of the scheduler are not interrupted, the time bounds L_S and U_S of actions of S are volatile. However, an execution of a process action may be pre-empted, for instance under a priority-based pre-emptive scheduler. Thus, the time bounds l and u for the actions in $\mathcal{G}(P)$ should be persistent in general. Moreover, in a concurrent program, a pre-empted action may be disabled by the execution of the actions of other processes. When the pre-empted process is resumed, this pre-empted (and disabled) action will not be executed and another enabled action in this process will be selected for execution. For this reason, we need the notion of a persistent timer.

Definition 14. *A* **persistent δ-timer** *t for an action τ in process p_i is defined as follows:*

$$\begin{aligned}
Persistent(t, \tau, \delta, \overline{v}) \stackrel{\Delta}{=} & t = \delta \wedge \\
& \Box[\quad (r(\tau) \wedge t' = now + \delta && \textit{taken} \\
& \quad \vee en(r(\tau)) \wedge \neg\tau \wedge t' = t && \textit{running} \\
& \quad \vee \neg en(\tau)' \wedge t' = now' + \delta && \textit{disabled} \\
& \quad \vee en(\tau) \wedge \neg run_i \wedge t' = t + (now' - now)) && \textit{pre-empted} \\
& \wedge ((\overline{v}, now)' \neq (\overline{v}, now))]_{(t, \overline{v}, now)}
\end{aligned}$$

Informally,

1. the persistent δ-timer t is initially (that is, when $now = 0$) set to δ;
2. it stays δ time units ahead of now as long as τ is not enabled (or equivalently speaking $\neg en(\tau)$ holds);
3. it remains unchanged during any time when τ is both enabled and run (that is, the enabling condition $en(r(\tau))$ holds) to record the execution time;
4. it is reset either just after a τ-step is taken or τ is disabled; and
5. it changes at the same rate as now when τ is enabled but not run (that is, $en(\tau) \wedge \neg run_i$ holds). This says that the time when a process is waiting for the processor or the execution of τ is pre-empted should not be counted as execution time.

Conditions (4) & (5) guarantee that timer t is persistent only when τ is pre-empted, and if τ is pre-empted the intermediate state at the point of pre-emption is not observable to other actions.

The conjunction of the defining formula of a persistent $u(\tau)$-timer T_τ for action τ and $MaxTime(u(\tau))$:

$$Persistent(T_\tau, \tau, u(\tau), \overline{v}) \wedge MaxTime(T_\tau)$$

is the specification of the upper persistent time bound condition for action $r(\tau)$, and this asserts that the τ-step of state transition must take place if the accumulated time when τ has been both enabled and run reaches $u(\tau)$. Similarly, the lower persistent time bound condition for action τ is specified by:

$$Persistent(t_\tau, \tau, l(\tau), \overline{v}) \wedge MinTime(t_\tau, r(\tau), \overline{v})$$

Notice that when there is no pre-emption in the execution of the program:

$$\Box(en(r(\tau)) \wedge \neg run'_i \Rightarrow (\neg en(\tau) \vee r(\tau)))$$

is ensured by the scheduler; the use of a persistent timer of τ in these two formulas is equivalent to the use of a volatile timer of $r(\tau)$:

1. $Persisten(t, \tau, \delta, \overline{v})$ initially sets t to δ, and keeps resetting t with $now + \delta$ as long as $\neg en(r(\tau))$. This is the same as in $Volatile(t, r(\tau), \delta, \overline{v})$ which sets t to ∞ and keeps it unchanged until $en(r(\tau))$ becomes $true$ and sets it to $now' + \delta$.
2. Assume $en(\tau) \wedge \neg run_i$ has been $true$ since, say $now = now_0$, and t was set by $Persisten(t, \tau, \delta, \overline{v})$ to $now_0 + \delta$. From now_0, $Persisten(t, \tau, \delta, \overline{v})$ increases t at the same rate by which now increases as there cannot be a pre-emption. This is the same as in $Volatile(t, r(\tau), \delta, \overline{v})$ where t was set to ∞ and kept unchanged unless run_i becomes $true$ when t is set to $now' + \delta$.

Thus, persistent timers allow the treatment of both pre-emptive and non-pre-emptive scheduling.

The specification of the timing condition for $\mathcal{G}(P)^T$ is defined as

$$B(\mathcal{G}(P)^T) \stackrel{\Delta}{=} \bigwedge_{\tau \in A} (Persistent(t_\tau, \tau, l(\tau), \overline{v}) \wedge MinTime(t_\tau, r(\tau), \overline{v})) \wedge \bigwedge_{\tau \in A} (Persistent(T_\tau, \tau, u(\tau), \overline{v}) \wedge MaxTime(T_\tau))$$

The exact specification of the timed scheduled program $\mathcal{I}(P^T)$ is

$$\begin{aligned} \Pi(\mathcal{I}(P^T)) &= \Pi(S) \wedge \Pi(\mathcal{G}(P)) \wedge RT \wedge B(S^T) \wedge B(\mathcal{G}(P)^T) \\ &= \Pi(S^T) \wedge \Pi(\mathcal{G}(P)^T) \end{aligned}$$

The correctness of the timed scheduled program $\mathcal{I}(P^T)$ is determined with respect to the specification of P^T, which does not refer to the variables $\overline{z}$ which are modified by the scheduler S. These variables (and those which are internal to S) are therefore hidden in the canonical specification

$$\Phi(\mathcal{I}(P^T)) \stackrel{\Delta}{=} \exists \overline{z} \cdot \Phi(S^T) \wedge \Phi(\mathcal{G}(P)^T)$$

We shall use this specification in the following section where we consider two ways of applying the transformational approach to real-time scheduling.

7.3 Reasoning About Scheduled Programs

Consider the implementation of a real-time program P^T using a real-time scheduler S^T which satisfies a property φ. Proof that this implementation satisfies a high-level timing property ψ, whose only free state variables are *now* and the external variables of S, can be used as the initial basis from which proofs of more detailed low level properties can later be established.

Because of the assumption that the program and the scheduler do not change the same variables, if S^T satisfies a property φ and $\varphi \wedge \Phi(\mathcal{G}(P)^T)$ implies ψ, then $\mathcal{I}(P^T)$ satisfies ψ. This is represented as the proof rule:

$$R1. \frac{\begin{array}{l} 1\ \Phi(S^T) \Rightarrow \varphi \\ 2\ \exists \overline{z} \cdot \varphi \wedge \Phi(\mathcal{G}(P)^T) \Rightarrow \psi \end{array}}{\Phi(\mathcal{I}(P^T)) \Rightarrow \psi}$$

Treating the effect of scheduling as a transformation of a program specification allows an abstract specification of a scheduler's *policy* to be used to prove the timing properties of the implementation of a real-time program.

7.4 Feasibility: Definition and Verification

Definition 15. *The timed scheduled program $\mathcal{I}(P^T)$ is* **feasible** *if the following holds: $\Phi(\mathcal{I}(P^T)) \Rightarrow \Phi(P^T)$. Therefore, if there is a refinement mapping by which the following implication can be proved:*

$$\Pi(\mathcal{I}(P^T)) \Rightarrow \widetilde{\Pi(P^T)}$$

Notice that the correctness of a scheduler is defined with respect to its specification (or its scheduling policy) while feasibility relates the specification of the program P^T to be scheduled to the specification of the scheduled program and requires the time bounds of all actions of the former to be met by the later. Assuming that $\Phi(S^T) \Rightarrow \varphi$, the feasibility of $\mathcal{I}(P^T)$ can be proved from Rule R1 as the implication:

$$\exists\, \overline{z} \cdot \varphi \wedge \Phi(\mathcal{G}(P)^T) \Rightarrow \Phi(P^T) \tag{1}$$

This formula can be manipulated in steps, using a refinement mapping.

Step 1. *Introduce auxiliary (dummy) timers into $\mathcal{G}(P)^T$ corresponding to the timers of P^T.*

This can be understood as allowing the scheduler to have a copy of the timers of $P^T = \langle P, L, U\rangle$. Define a set of auxiliary variables:

$$\mathit{dummies} \triangleq \{\, h_\tau, H_\tau \;\mid\; \tau \in A \,\}$$

where h_τ and H_τ are respectively defined by the formulas $\mathit{Volatile}(h_\tau, \tau, L(\tau), \overline{v})$ and $\mathit{Volatile}(H_\tau, \tau, U(\tau), \overline{v})$. Let:

$$D(\mathit{dummies}) \triangleq \bigwedge_{\tau \in A} \mathit{Volatile}(h_\tau, \tau, L(\tau), \overline{v}) \wedge \mathit{Volatile}(H_\tau, \tau, U(\tau), \overline{v})$$

Then (1) is equivalent to:

$$\exists\, \mathit{dummies}, \overline{z} \cdot \varphi \wedge \Phi(\mathcal{G}(P)^T) \wedge D(\mathit{dummies}) \Rightarrow \Phi(P^T) \tag{2}$$

Step 2. *Define the refinement mapping.*

Recall that the internal variables of P are assumed to be $\overline{x}$. A *refinement mapping* from the states over $\overline{x} \cup \overline{z} \cup \mathit{timer}(\mathcal{G}(P)^T) \cup \mathit{dummies}$ to the states over $\overline{x} \cup \mathit{timer}(P^T)$ is defined as follows:

$$\tilde{y} = \begin{cases} h_\tau \text{ if } y \text{ is a timer } t_\tau \in \mathit{timer}(P^T) \\ H_\tau \text{if } y \text{ is a timer } T_\tau \in \mathit{timer}(P^T) \\ y \;\text{ if } y \in \overline{x} \end{cases}$$

Let $\mathit{TimedSched} \triangleq \varphi \wedge \Pi(\mathcal{G}(P)^T) \wedge D(\mathit{dummies})$. Then (2) can be proved by proving:

$$\mathit{TimedSched} \Rightarrow \widetilde{\Pi(P^T)} \tag{3}$$

Step 3. *Discard identical substitutions.*

Recall that $\widetilde{\Pi(P^T)} = \widetilde{\Pi(P)} \wedge \widetilde{RT} \wedge \widetilde{B(P^T)}$. Obviously, we have that $\widetilde{RT} = RT$ and $\widetilde{\Pi(P)} = \Pi(P)$. Also $\Pi(\mathcal{G}(P)^T)$ implies $\Pi(\mathcal{G}(P))$, which in turn implies $\Pi(P)$. Therefore, $\widetilde{RT}$ and $\widetilde{\Pi(P)}$ can be discarded from the right hand side of the implication in (3).

$$\mathit{TimedSched} \Rightarrow \widetilde{B(P^T)} \tag{4}$$

Step 4. *Discard the actions on timers.*

$$\begin{aligned}\widetilde{B(P^T)} = & \bigwedge_{\tau \in A} \mathit{Volatile}(h_\tau, \tau, L(\tau), \overline{v}) \wedge \mathit{MinTime}(h_\tau, \tau, \overline{v}) \wedge \\ & \mathit{Volatile}(H_\tau, \tau, U(\tau), \overline{v}) \wedge \mathit{MaxTime}(H_\tau) \\ = & D(\mathit{dummies}) \wedge \bigwedge_{\tau \in A} (\mathit{MaxTime}(H_\tau) \wedge (\mathit{MinTime}(h_\tau, \tau, \overline{v}))\end{aligned}$$

Since $D(\mathit{dummies})$ appears on the left hand side of (4), what remains to be proved is that the following implication holds for each action τ of P.

$$\mathit{TimedSched} \Rightarrow \mathit{MaxTime}(H_\tau) \wedge \mathit{MinTime}(h_\tau, \tau, \overline{v}) \tag{5}$$

7.5 Proof Rules for Feasibility

Implication 5 suggests that the feasibility of an implementation of a real-time program P^T can be proved using the following rule:

$$R2.\ \frac{\begin{array}{l}1\ \Phi(S^T) \Rightarrow \varphi \\ 2\ \mathit{TimedSched} \Rightarrow \mathit{MaxTime}(H_\tau) \text{ for } \tau \in A \\ 3\ \mathit{TimedSched} \Rightarrow \mathit{MinTime}(h_\tau, \tau, \overline{v}) \text{ for } \tau \in A\end{array}}{\Phi(\mathcal{I}(P^T)) \Rightarrow \Phi(P^T)}$$

Notice that both $\mathit{MaxTime}(H_\tau)$ and $\mathit{MinTime}(h_\tau, \tau, \overline{v})$ contain primed state variables. Therefore, rules for proving invariant properties cannot be used directly to establish the premises (2) and (3) in Rule R2. We provide two rules for introducing invariants.

To prove premise (2) in Rule R2, we have the following rule:

$$R3.\ \frac{\begin{array}{l}1\ \Phi(S^T) \Rightarrow \varphi \\ 2\ \mathit{TimedSched} \Rightarrow \Box(\mathit{en}(\tau) \Rightarrow T_\tau \leq H_\tau) \text{ for } \tau \in A\end{array}}{\mathit{TimedSched} \Rightarrow \mathit{MaxTime}(H_\tau) \text{ for } \tau \in A}$$

By symmetry, for premise (3) in Rule R2:

$$R4.\ \frac{\begin{array}{l}1\ \Phi(S^T) \Rightarrow \varphi \\ 2\ \mathit{TimedSched} \Rightarrow \Box(\mathit{run}_i \Rightarrow t_\tau \geq h_\tau) \text{ for } \tau \text{ in } p_i\end{array}}{\mathit{TimedSched} \Rightarrow \mathit{MinTime}(h_\tau, \tau, \overline{v}) \text{ for } \tau \in A}$$

TimedSched can be converted into a normal form as the conjunction of a safety property and a liveness property:

$$\exists\, \overline{x} \cdot \Theta \wedge \Box[\mathcal{N}]_{\overline{y}} \wedge \mathcal{L}$$

where $\overline{x}$ and $\overline{y}$ are sets of variables, Θ is a state predicate, $\mathcal{N}$ is an action and $\mathcal{L}$ is the time divergence property, $\forall\, t \cdot \Diamond(\mathit{now} > t)$.

Let this formula be denoted by Ω. An invariant Q of Ω can be proved using the rule:

$$R5.\ \frac{\begin{array}{ll}1\ \Theta \Rightarrow Q & \text{Initially } Q \text{ holds} \\ 2\ Q \wedge \mathcal{N} \Rightarrow Q' & \text{Each step of the transition preserves } Q\end{array}}{\Omega \Rightarrow \Box Q}$$

7.6 Feasibility of Fault-Tolerant Real-Time Programs

The occurrence of a fault-action does not depend on the scheduler and the F-affected scheduled program of P^T by a scheduler S is modelled as $\mathcal{F}(\mathcal{I}(P^T), F)$ whose exact specification is:

$$\Pi(\mathcal{F}(\mathcal{I}(P^T), F)) = \Pi(S) \wedge \Pi(\mathcal{F}(\mathcal{G}(P), F)) \wedge RT \wedge B(S^T) \wedge B(\mathcal{G}(P^T))$$

Let *TimedSched* (from the previous subsection) be redefined as:

$$TimedSched \stackrel{\Delta}{=} \varphi \wedge \Pi(\mathcal{F}(\mathcal{G}(P^T), F)) \wedge D(dummies)$$

Definition 16. *Taking the same set of dummy variables dummies and the refinement mapping from the previous subsection, the implementation* $\mathcal{I}(P^T)$ *is* F**-tolerantly feasible** *if the following implication holds:*

$$TimedSched \Rightarrow \Pi(\widetilde{\mathcal{F}(P^T}, F))$$

Then all the equations and rules in the previous section remain valid for fault-tolerant feasibility.

Assume that a real-time program $P^T = \langle P, L, U \rangle$ is a F-tolerant refinement of a program P_h^T for a given set F of fault-actions. Then any F-tolerant feasible implementation of P^T is a F-tolerant refinement of P_h^T.

This assumes that the execution of the scheduler is not faulty, and that F-tolerance is provided by the program to be scheduled. It is also possible for a non-fault-tolerant program to be executed under a specially designed scheduler so that the implementation of the faulty program is fault-tolerant [164].

For example, a scheduler can be designed to tolerate processor failures. Assume each process of P^T keeps taking checkpoints of its local states by a transformation $\mathcal{C}(P^T)$. We add recovery process(es) to $\mathcal{C}(P^T)$ by a transformation $\mathcal{R}(\mathcal{C}(P^T))$. Faults and their effects on processes are modelled as before. The implementation transformation $\mathcal{I}$ is applied to $\mathcal{F}(\mathcal{R}(\mathcal{C}(P^T)), F)$. When a processor fails, the scheduler must submit the recovery process to a non-failed processor. If the processors are *fail-stop*, no checkpointing or recovery may be needed. The scheduler only needs to re-schedule a process executing on a failed processor to a non-failed processor, where that is possible. Our theory and verification techniques also apply to formal reasoning about such fault-tolerant implementations.

7.7 Scheduling Open Systems

In the model of programs given so far, we have assumed that a real-time program implements the specification of a *closed system*: values are supplied to the program through the initial values of variables or by executing a nondeterministic *input* operation.

In many cases, a program is linked to an external environment from which it receives data and to which it must send responses. The appearance of the inputs often follows a timing pattern, for example with *periodic* or *aperiodic* repetition.

Definition 17. *An* **open system** *is a pair $O = (E, P)$ consisting of a program P which interacts with an environment E. The set $\overline{v}_o$ of variables of O is the union of the sets $\overline{x}$ and $\overline{y}$ of* local variables *of P and E and the set $\overline{v}$ of* interface variables *through which P and E interact.*

Let program P consist of an initial predicate $\Theta_{\overline{x}}$ over its local variables $\overline{x}$ and a set of atomic actions on the program variables $\overline{v}_p = \overline{x} \cup \overline{v}$. Let the environment E consist of an initial predicate Θ over the environment variables $\overline{v}_e = \overline{y} \cup \overline{v}$ and a set of atomic actions on the variables $\overline{v}_e$.

Let ν be an action formula that defines the state transitions by which P changes the values of the interface variables. It is then required [2] that:

$$\mathcal{N}_P \Rightarrow \nu \vee (\overline{v}' = \overline{v}) \text{and} \mathcal{N}_E \Rightarrow \neg\nu \vee (\overline{v}' = \overline{v})$$

where $\mathcal{N}_P$ is the next-state relation of P.

As before, we define:

$$\Pi(P) \stackrel{\Delta}{=} \Theta_{\overline{x}} \wedge \Box[\neg\nu \wedge (\overline{x}' = \overline{x}) \vee \mathcal{N}_P]_{\overline{v}_p} \text{and} \Phi(P) \stackrel{\Delta}{=} \exists\, \overline{x}.\Pi(P)$$

$$\Pi(E) \stackrel{\Delta}{=} \Theta \wedge \Box[\nu \wedge (\overline{y}' = \overline{y}) \vee \mathcal{N}_E]_{\overline{v}_e} \quad \text{and} \Phi(E) \stackrel{\Delta}{=} \exists\, \overline{y} \cdot \Pi(E)$$

The specification $\Phi(O)$ of an open system $O = (E, P)$ then defines the condition under which the system guarantees the property $\Phi(P)$ if the environment satisfies the assumption $\Phi(E)$.

$$\Phi(O) \stackrel{\Delta}{=} \Phi(E) \Rightarrow \Phi(P)$$

The conjunction $\Phi(E) \wedge \Phi(P)$ describes the closed system consisting of P and its environment E and is:

$$\exists\, \overline{x}, \overline{y} \cdot \Theta \wedge \Theta_{\overline{x}} \wedge \Box[\mathcal{N}_P \vee \mathcal{N}_E]_{\overline{v}_o}$$

Program P_l refines (or implements) a program P_h in environment E iff:

$$(\Phi(E) \Rightarrow \Phi(P_l)) \Rightarrow (\Phi(E) \Rightarrow \Phi(P_h))$$

and this reduces to:

$$\Phi(E) \wedge \Phi(P_l) \Rightarrow \Phi(P_h)$$

The program and its environment can be treated as the real-time programs $P^T = \langle P, L, U \rangle$ and $E^T = \langle E, L_e, U_e \rangle$ respectively. Since time is global, it need not be advanced by both of them. We choose to let the program advance time and define:

$$\Phi(E^T) \stackrel{\Delta}{=} \exists\, \overline{y}, \mathit{timer}(E^T) \cdot \Pi(E) \wedge B(E^T)$$

The *real-time open system* $O^T = (E^T, P^T)$ is specified by

$$\Phi(O^T) \stackrel{\Delta}{=} \Phi(E^T) \Rightarrow \Phi(P^T)$$

We would like to point out that the canonical form of an open real-time specification given here is simpler than that in [2] but is sufficient for our purposes as we shall not be considering the problem of composing open systems.

A real-time property φ of an open system $O^T = (E^T, P^T)$ states that program P^T guarantees the property φ under the environment assumption E^T. This requires proving the implication:

$$\Phi(O^T) \Rightarrow (\Phi(E^T) \Rightarrow \varphi)$$

or, equivalently,

$$\Phi(E^T) \wedge \Phi(P^T) \Rightarrow \varphi$$

In a real-time environment E^T, implementation of a real-time program P^T by a scheduler S^T on a set of processors can be described by transformation $\mathcal{I}(O^T) \stackrel{\Delta}{=} (E^T, \mathcal{I}(P^T))$, in which $\mathcal{I}(P^T)$ is as defined in Section 4.2 for a closed system, and $\overline{z}$ denotes the variables that may be changed by the scheduler.

The feasibility of the implementation relies on proving the refinement relation: $O^T \sqsubseteq \mathcal{I}(O^T)$ Notice that this refinement relation is equivalent to the implication

$$\Phi(\mathcal{I}(O^T)) \Rightarrow \Phi(O^T)$$

Equivalently this can be proved by proving:

$$\Phi(E^T) \wedge \Phi(\mathcal{I}(P^T)) \Rightarrow \Phi(P^T) \tag{6}$$

If we use this definition of schedulability for open systems in our derivation of Rules R1–R2, we can show that these rules are also valid for open systems.

7.8 Running Example Continued

In the timed fault-tolerance processor-memory interface program P_3^T, let RW_3^p be an environment action with its lower and upper bounds (that equal to the period) set to ρ. Partition the remaining actions into four processes:

$$\begin{array}{llll}
p_4 = \{Vote\} & p_i = \{R_3^{m_i}, W_3^{m_i}\} & & \text{for } i = 1, 2, 3 \\
L_3(Vote) = 0 & L_3(R_3^{m_i}) = L_3(W_3^{m_i}) = 0 & & \text{for } i = 1, 2, 3 \\
U_3(R_3^{m_i}) = D_{r_i} & U_3(W_3^{m_i}) = D_{w_i} \leq D_2 & & \text{for } i = 1, 2, 3 \\
U_3(Vote) = D_{vote} & D_{r_i} + D_{vote} \leq D_2 & & \text{for } i = 1, 2, 3
\end{array}$$

where D_2 is the deadline of the memory actions in the real-time interface program P_2^T implemented by P_3^T.

Let the memory processes be implemented on a single processor using a nondeterministic scheduler. Ignore the details of the scheduler program: for instance, we can assume that it randomly chooses an enabled process. If there is no

overhead in the scheduling, the scheduler can be specified as a real-time program $S^T = \langle S, L, U \rangle$:

$$\begin{aligned}
\overline{z} &\triangleq \{\ run_i \mid i = 1, 2, 3, 4\ \} \\
\Theta &\triangleq idle \\
g_i &\triangleq true \textbf{ if } \text{an action of } p_i \text{ is enabled } \textbf{else } false,\ i = 1, 2, 3, 4 \\
sch &\triangleq \bigvee_{i=1}^{4} (g_i \wedge (idle \vee \neg g_{i\oplus 1} \wedge \neg g_{\oplus 2}) \wedge run_i') \vee (\bigwedge_{i=1}^{4} \neg g_i) \wedge idle' \\
U(sch) &= 0
\end{aligned}$$

Now assume that the computation times for the actions of the processes satisfy the following condition:

$$\begin{aligned}
& l(Vote) = l(R_3^{m_i}) = l(W_3^{m_i}) = 0 && \text{for } i = 1, 2, 3 \\
& u(W_3^{m_1}) + u(W_3^{m_2}) + u(W_3^{m_3}) \leq min\{D_{w_i}\} && \text{for } i = 1, 2, 3 \\
& u(R_3^{m_1}) + u(R_3^{m_2}) + u(R_3^{m_3}) \leq min\{D_{r_i}\} && \text{for } i = 1, 2, 3 \\
& u(Vote) \leq D_{vote}
\end{aligned}$$

Then it can be proved using the rules in Section 7.4 that the implementation of P_3^T by the scheduler S^T on the given processor is F_2-fault-tolerantly feasible.

Intuitively, the processor actions ensure that *read* and *write* tasks do not arrive at the same time. Once a *write* or a *read* operation is issued, all the three *write* or *read* tasks are enabled in the three processes. The scheduler selects one process at a time to execute until all of them are executed; in total, this takes at most the sum of the computation times of the three tasks. The *Vote* process p_4 can be ready only when the other processes are not ready.

Proof (sketch of F_2-tolerant feasibility). Rule 3 in Section 7.5 requires that we prove that the following predicates are invariants of the F_2-implementation:

$$\begin{aligned}
& I_R^i \triangleq op_i = r \Rightarrow T_{R_3^{m_i}} \leq H_{R_3^{m_i}} && \text{for } i = 1, 2, 3 \\
& I_W^i \triangleq op_i = w \Rightarrow T_{W_3^{m_i}} \leq H_{W_3^{m_i}} && \text{for } i = 1, 2, 3 \\
& I_V \triangleq v_1 \wedge v_2 \wedge v_3 \Rightarrow T_{Vote} \leq H_{Vote}
\end{aligned}$$

The proofs of these invariants are very similar. We present only a sketch of the proof for I_R^1. Let u_i be used for $u(R_3^{m_i})$, H_i for $H_{R_3^{m_i}}$, and T_i for $T_{R_3^{m_i}}$, $i = 1, 2, 3$.

In general, it may not always possible to prove an invariant Q directly from Rule *R5* in Section 7.5. Instead, we have to use this rule to prove a stronger invariant which implies Q. To prove that I_R^1 is an invariant, prove the following invariants I_1–I_7, the conjunction of which is an invariant and implies I_R^1:

$$I_1 \triangleq (\bigwedge_{i=1}^{3} (op_i = r \wedge \neg run_i)) \Rightarrow (\bigwedge_{i=1}^{3} (H_i = now + D_{r_i}) \wedge (T_j = now + u_j))$$

Notice that:

$$I_1 \Rightarrow ((\bigwedge_{i=1}^{3} (op_i = r \wedge \neg run_i)) \Rightarrow (\bigwedge_{k\neq i\neq j\neq k} (H_i - T_i \geq u_j + u_k))) \qquad (7)$$

Informally, I_1 is an invariant because (a) the timers H_i and T_i are respectively set with $now + D_{r_i}$ and $now + u_i$ when $op_i = r$ is changed from *false* to *true* and (b) there is no overhead in the scheduling and thus *now* cannot advance before one of the three ready processes is scheduled for execution.

If after a read operation is issued, the scheduler chooses process p_2 first for execution, we have the following invariant.

$$I_2 \stackrel{\Delta}{=} (\bigwedge_{i=1}^{3} (op_i = r) \wedge run_2) \Rightarrow H_1 - T_1 \geq u_3 + T_2 - now$$

The proof of this invariant uses invariant I_1 and its implication 7 together with the following two facts:

1. A transition from a non-$\bigwedge_{i=1}^{3} (op_i = r) \wedge run_2$-state to a $\bigwedge_{i=1}^{3} (op_i = r) \wedge run_2$-state can only be a transition from a $\bigwedge_{i=1}^{3} (op_i = r \wedge \neg run_i)$-state and carried out by an scheduling action. This scheduling action does not change *now* and the timers. Formally, let $\mathcal{N}_1$ be this action:

   $$\mathcal{N}_1 \stackrel{\Delta}{=} (\bigwedge_{i=1}^{3} (op_i = r \wedge \neg run_i)) \wedge run'_2$$

   By I_1 and implication (7), we have:

   $$\begin{aligned} \mathcal{N}_1 \Rightarrow\ & (T_2 = now + u_2) \wedge (now, H_1 - T_1 \geq u_3 + u_2) \\ & \wedge unchanged(H_1, T_1, T_2) \end{aligned}$$

   Hence,

   $$\mathcal{N}_1 \Rightarrow (H'_1 - T'_1 \geq u_3 + T'_2 - now)$$

2. The amount of time for which $\bigwedge_{i=1}^{3} (op_i = r) \wedge run_2$ has remained *true* up to *now* is the time $u_2 - (T_2 - now)$ spent on the execution of $R_3^{m_2}$ that has been added to T_1 as it has been persistent. The only action which may falsify I_2 is:

   $$\begin{aligned} \mathcal{N}_2 \stackrel{\Delta}{=}\ & (op_1 = r) \wedge (op_2 = r) \wedge (op_3 = r) \wedge run_2 \\ & \wedge (now' > now) \wedge (T'_1 = T_1 + (now' - now)) \\ & \wedge (T'_2 = T_2) \wedge (H'_1 = H_1) \end{aligned}$$

   where we ignore the changes in other variables which are irrelevant to I_2. Clearly $I_2 \wedge \mathcal{N}_2 \Rightarrow I'_2$ as:

   $$\begin{aligned} H'_1 - T'_1 =& H_1 - T_1 - now' + now \\ \geq& u_3 + T_2 - now - now' + now \\ =& u_3 + T'_2 - now' \end{aligned}$$

Similar to I_2, we have the following invariant if the scheduler chooses p_3 first for execution:

$$I_3 \stackrel{\Delta}{=} (\bigwedge_{i=1}^{3} (op_i = r) \wedge run_3) \Rightarrow H_1 - T_1 \geq u_2 + T_3 - now$$

If the scheduler chooses p_1 for execution first, then:

$$I_4 \stackrel{\Delta}{=} (\bigwedge_{i=1}^{3} (op_i = r) \wedge run_1) \Rightarrow H_1 - T_1 \geq u_1 + u_3$$

These four invariants consider the cases when none of the three processes has completed the issued read operation. We have the following three invariants about the cases when one of or both of p_2 and p_3 have completed the operation.

If p_2 has completed the operation, we have the invariant:

$$I_5 \stackrel{\Delta}{=} (op_1 = r) \wedge (op_2 \neq r) \wedge (op_3 = r) \wedge \neg run_1 \Rightarrow H_1 - T_1 \geq T_3 - now$$

This characterizes the fact that the time spent on the whole execution of $R_3^{m_2}$ and on the partial execution of $R_3^{m_3}$ has been added to T_1. The proof of this invariant uses I_2. Similarly, if p_2 has completed the operation we have the invariant:

$$I_6 \stackrel{\Delta}{=} (op_1 = r) \wedge (op_2 = r) \wedge (op_3 \neq r) \wedge \neg run_1 \Rightarrow H_1 - T_1 \geq T_2 - now$$

Finally, we have the invariant:

$$I_7 \stackrel{\Delta}{=} (op_1 = r) \wedge ((op_2 \neq r) \vee (op_3 \neq r)) \Rightarrow H_1 - T_1 \geq 0$$

This characterizes the fact that the time spent on the execution of one or both of $R_3^{m_2}$ and $R_3^{m_3}$ has been added to T_1.

The nondeterministic scheduler can be refined to a deterministic one by assigning priorities to the processes. For example, let process p_i have a higher priority than p_j if $i < j$. Modify the action *sch* of the scheduler into sch_1 such that the process with the highest priority among the ready processes is scheduled for execution but no pre-emption is allowed:

$$\begin{aligned} sch_1 \stackrel{\Delta}{=}\ & idle \wedge g_1 \wedge run_1' \\ & \vee idle \wedge \neg g_1 \wedge g_2 \wedge run_2' \\ & \vee idle \wedge \neg g_1 \wedge \neg g_2 \wedge g_3 \wedge run_3' \\ & \vee idle \wedge \neg g_1 \wedge \neg g_2 \wedge \neg g_3 \wedge g_4 \wedge run_4' \\ & \vee \bigvee_{i=1}^{4} (run_i \wedge \neg g_i \wedge idle') \end{aligned}$$

Then the modified scheduler also gives a feasible F_2-tolerant implementation of P_3^T on the given processor, as the new action sch_1 action implies the old *sch*.

7.9 Fixed Priority Scheduling with Pre-emption

The techniques presented in the previous subsections can be used to produce results similar to those obtained using scheduling theory. We demonstrate this by proving the feasibility condition given in [40] for implementing a set of independent tasks using fixed priority scheduling with pre-emption.

Consider an open system $O = (E, P)$ where program P consists of n independent processes (or tasks) which are represented by the atomic actions $\tau_1, \ldots, \tau_n$. The environment E is used to represent the actions of releasing (or invoking, or activating) the tasks periodically. In general, these actions may be clock events or external events to which the processes need to respond. Let ρ_i be the period of τ_i, for $i = 1, \ldots, n$.

7.10 Specification of the Program

To specify the system in TLA, let inv_i and com_i be integer variables representing the number of invocations and completions of each task i. Then the specification of the real-time system $O^T = (E^T, P^T)$ can be given as: for $i = 1, \ldots, n$

$$
\begin{array}{lll}
\Theta & \triangleq (0 \leq inv_i \leq 1) \wedge (com_i = 0) & \\
\alpha & \triangleq inv_i' = inv_i + 1 & \text{action of } E \text{ for task invocation} \\
\tau_i & \triangleq inv_i > com_i \wedge com_i' = com_i + 1 & \text{action of } P \text{ for task completion} \\
\nu & \triangleq \bigvee_{i=1}^{n} (com_i' = com_i + 1) & \\
L(\alpha_i) = U(\alpha_i) = \rho_i & & \text{period of invocation} \\
L(\tau_i) = 0 \text{ and } U(\tau_i) = D_i & & \text{deadline of task}
\end{array}
$$

A basic (functional) requirement for the system is that each invocation of a task is completed before its next invocation, that is,

$$\Phi(E^T) \wedge \Phi(P^T) \Rightarrow \bigwedge_{i=1}^{i=n} \Box(inv_i \geq com_i \geq inv_i - 1)$$

From the rules for proving an invariant in TLA, this implication holds if $D_i < \rho_i$. It must now be shown that an implementation of the program P^T on a uniprocessor system is feasible.

7.11 Specification of the Scheduling Policy

Let the system be implemented on a single processor using a pre-emptive, fixed-priority scheduler; assume that there is no scheduling overhead. Let τ_i have a higher priority than τ_j if $i < j$. Let g_i denote the enabling condition of task τ_i, and hr_i assert that τ_i has the highest priority among the current enabled (or ready) tasks:

$$g_i \triangleq inv_i > com_i hr_i \triangleq g_i \wedge \forall j < i \cdot \neg g_j$$

Then the scheduler, denoted by $S^T = \langle S, L, U \rangle$, can be specified as follows:

$$\begin{aligned} sch_i &\stackrel{\Delta}{=} idle \wedge hr_i \wedge run'_i \vee && \text{higher task runs first} \\ &\quad \exists j \neq i \cdot (run_j \wedge hr_i \wedge run'_i \wedge \neg run'_j) && \text{higher task pre-empts lower task} \\ \mathcal{N}_S &= \bigvee_{i=1}^{n} sch_i \\ U(sch_i) &= L(sch_i) = 0 && \text{no overhead} \end{aligned}$$

Specifications of various scheduling policies can be found in [169].

According to S^T, at any time at most one process is running on the processor:

$$Valid \stackrel{\Delta}{=} \Box(i \neq j \Rightarrow \neg(run_i \wedge run_j))$$

7.12 Feasibility

Let the computation time for each task τ_i be in the interval $[0, C_i]$, that is, $l(\tau_i) = 0$ and $u(\tau_i) = C_i$. Assume ρ_i, D_i and C_i are non-negative integers for $i = 1, \ldots, n$. The worst-case response time (or completion time) R_i for each task τ_i can be defined as a recursive equation [136]. We shall instead use the equivalent recurrence relation defined in [40]. The $(n + 1)$th response time $R_i^{(n+1)}$ for process i is:

$$R_i^{(n+1)} = C_i + \sum_{j=1}^{i-1} \lceil \frac{R_i^{(n)}}{\rho_j} \rceil \times C_j \tag{8}$$

If $R_i^{(0)}$ is initially set C_i, and:

$$R_i = \lim_{n \to \infty} R_i^{(n)}$$

scheduling theory shows that:

> the implementation of the program by the scheduler on the given processor is *feasible* iff $R_i \leq D_i$, for $i = 1, \ldots, n$.

This condition can be shown to be necessary by finding an execution in which a task misses its deadline if the condition does not hold. However, to prove formally that the condition is sufficient, we need to prove the following refinement.

Theorem 1. *For the given program, $O^T = (E^T, P^T)$, the scheduler, S^T, and the processor*

$$O^T \sqsubseteq \mathcal{I}(O^T)$$

provided $R_i \leq D_i$ for $i = 1, \ldots, n$.

By Implication (6) in Section 7.7, this is equivalent to showing that the following holds:

$$\Pi(E^T) \wedge \Pi(S^T) \wedge \Pi(\mathcal{G}(P^T)) \wedge D(dummies) \Rightarrow \widetilde{\Pi(P^T)} \tag{9}$$

where $D(dummies)$ and refinement mapping are as defined in Section 5.1.

Before proving (9), let us discuss how the persistent timer T_{τ_i} is used to predict the completion time of an invocation of task τ_i by considering its first invocation.

As a special case, consider any time *now* before the completion of the *first* invocation of task τ_i (that is, when $com_i = 0$ and $inv_i > 0$). Assume all tasks τ_j, $j = 1, \ldots, i-1$, with higher priorities than τ_i have met their deadlines so far. Then, in the worst case, when all tasks τ_j use C_j units of computation time, the time spent up to *now* on executing higher priority processes is:

$$Comp(i, now) \stackrel{\Delta}{=} \sum_{j=1}^{i-1} com_j \times C_j + \sum_{j=1}^{i-1} (inv_j - com_j) \times (C_j - (T_{\tau_j} - now)) \tag{10}$$

where $C_j - (T_{\tau_j} - now)$ is the time spent so far on the last invocation of τ_j. Thus (10) becomes:

$$Comp(i, now) = \sum_{j=1}^{i-1} inv_j \times C_j - \sum_{j=1}^{i-1} (inv_j - com_j) \times (T_{\tau_j} - now) \tag{11}$$

Assume δ is the time already spent on τ_i up to *now*. Then:

$$now = Comp(i, now) + \delta$$

As T_{τ_i} has been persistent during the time when tasks of higher priorities are being executed, we have

$$T_{\tau_i} = Comp(i, now) + C_i$$

Thus, $T_{\tau_i} = now + (C_i - \delta)$ *predicts* that the cumulative time needed to complete τ_i after *now* will not exceed $C_i - \delta$; this time may be divided into smaller units whose sum is T_{τ_i}. For the first invocation of τ_i to be completed before its deadline, T_{τ_i} should never exceed H_{τ_i} (which is always equal to D_i before the completion of τ_i).

Thus, we need to prove that the left hand side (or LHS) of Implication (9) has the following predicate as an invariant:

$$(com_i = 0 \wedge inv_i > com_i) \Rightarrow T_{\tau_i} \leq C_i + Comp(i, now)$$

In general, at any time before an invocation of τ_i is completed, $H_{\tau_i} - D_i$ records the time t_0 (that is, the value of *now* at that time) of the current invocation of τ_i: at that time H_{τ_i} was $t_0 + D_i$ and it has remained unchanged as τ has not been completed. The definition of the longest possible time, $Comp(i, now)$, spent on

executing tasks with priorities higher than that of τ_i's defined by Equation (11) can be generalized as:

$$Comp(i, now) \stackrel{\Delta}{=} \sum_{j=1}^{i-1} \lceil \frac{inv_j \times \rho_j - (H_{\tau_i} - D_i)}{\rho_j} \rceil \times C_j - \sum_{j=1}^{i-1} (inv_j - com_j) \times (T_{\tau_j} - now)$$

This leads to the following lemma which implies Theorem 1.

Lemma 1. *LHS(9) has the following invariants: for $i = 1, \ldots, n$*

$$I_{1i} \stackrel{\Delta}{=} (com_i < inv_i) \Rightarrow T_{\tau_i} \leq C_i + Comp(i, now) + (H_{\tau_i} - D_i)$$
$$I_{2i} \stackrel{\Delta}{=} (com_i < inv_i) \Rightarrow Comp(i, now) \leq \sum_{j=1}^{i-1} \lceil (now - (H_{\tau_i} - D_i))/\rho_j \rceil \times C_j$$
$$I_{3i} \stackrel{\Delta}{=} (com_i < inv_i) \Rightarrow C_i + Comp(i, now) \leq R_i$$
$$I_{4i} \stackrel{\Delta}{=} inv_i - 1 \leq com_i \leq inv_i$$

Proof (of Lemma 1). The proof follows the general routine of proving invariants by showing that each of the I's holds initially and is preserved by each allowed state transition in the program.

It is easy to check that these invariants hold for $i = 1$. Assume that they hold for some $i - 1$, where $i \geq 1$. We prove they hold for i.

Take the case when $H_{\tau_i} = D_i$ for the first invocation of τ_i, that is the execution of the first invocation of τ_i. (The proof of the general case is very similar.)

For the special case, the lemma is rewritten as follows:

$$I_{1i} \stackrel{\Delta}{=} (com_i = 0) \wedge (inv_i > 0) \Rightarrow T_{\tau_i} \leq C_i + Comp(i, now)$$
$$I_{2i} \stackrel{\Delta}{=} (com_i = 0) \wedge (inv_i > 0) \Rightarrow Comp(i, now) \leq \sum_{j=1}^{i-1} \lceil now/\rho_j \rceil \times C_j$$
$$I_{3i} \stackrel{\Delta}{=} (com_i = 0) \wedge (inv_i > 0) \Rightarrow C_i + Comp(i, now) \leq R_i$$
$$I_{4i} \stackrel{\Delta}{=} inv_i - 1 \leq com_i \leq inv_i$$

where:

$$Comp(i, now) = \sum_{j=1}^{i-1} inv_j \times C_j - \sum_{j=1}^{i-1} (inv_j - com_j) \times (T_{\tau_j} - now)$$

Initially, I_{1i} holds as $T_{\tau_i} = C_i$. We analyze all possible state transitions allowed by $LHS(9)$ that may change the states of variables in $Comp(i, now)$.

Case 1: For $j = 1, \ldots, i - 1$, let

$$A_{1j} \stackrel{\Delta}{=} com_i = 0 \wedge inv'_j = inv_j + 1$$

In this case, I'_{1i} is

$$com'_i = 0 \Rightarrow T_{\tau_i} \leq C_i + Comp(i, now) + C_j - (T_{\tau_j} - now)$$

It is easy to prove that $LHS(9) \Rightarrow \Box(C_j - (T_{\tau_j} - now > 0))$. Thus,

$$I_{1i} \wedge A_{1j} \Rightarrow I'_{1i}$$

Case 2: For $j = 1, \ldots, i-1$, consider

$$A_{2j} \stackrel{\Delta}{=} com_i = 0 \wedge com'_j = com_j + 1 \wedge (T'_{\tau_j} = now + C_j)$$

By the induction assumption that $\Box(inv_j - 1 \leq com_j \leq inv_j)$, we know that I'_{1i} is equal to

$$com'_i = 0 \Rightarrow T_{\tau_i} \leq C_i + \sum_{j=1}^{i-1} inv_j \times C_j - \sum_{i>k\neq j} (inv_k - com_k) \times (T_{\tau_k} - now)$$

Thus, $I_{1i} \wedge A_{2j} \Rightarrow I'_{1i}$ holds.

Case 3: For $j = 1, \ldots, i-1$, define

$$A_{3j} \stackrel{\Delta}{=} com_i = 0 \wedge (com_j < inv_j) \wedge run_j \wedge (now' > now) \wedge T'_{\tau_i} = T_{\tau_i} + (now' - now) \wedge \varphi$$

where

$$\varphi \stackrel{\Delta}{=} \bigwedge_{i>k\neq j} (com_k < inv_k \Leftrightarrow (T'_{\tau_k} = T_{\tau_k} + (now' - now)))$$

Note that $(inv_k - com_k) = 0$ iff $\neg(com_k < inv_k)$ by the induction assumption for $k < i$. Thus, I'_{1i} becomes

$$com_i = 0 \Rightarrow T_{\tau_i} + (now' - now) \leq C_i + \sum_{k=1}^{i-1} inv_k \times C_k - \sum_{i>k\neq j} (inv_k - com_k) \times (T_{\tau_k} - now) - (T_{\tau_j} - now')$$

This is the same as

$$com_i = 0 \Rightarrow T_{\tau_i} \leq C_i + \sum_{k=1}^{i-1} inv_k \times C_k - \sum_{k=1}^{i-1} (inv_k - com_k) \times (T_{\tau_k} - now)$$

Thus, $I_{1i} \wedge A_{3j} \Rightarrow I'_{1i}$ holds.

Case 4: Finally consider:

$$A_4 \stackrel{\Delta}{=} (com_i = 0) \wedge run_i \wedge \varphi \wedge (now' > now)$$

where φ is defined as in **Case 3**, except for j being taken into account. Then, the same argument as in **Case 3** leads to I'_{1i} becoming:

$$com_i = 0 \Rightarrow T_{\tau_i} \leq C_i + \sum_{j=1}^{i-1} inv_j \times C_j - \sum_{j=1}^{i-1} (inv_j - com_j) \times (T_{\tau_j} - now)$$

And $I_{1i} \wedge A_4 \Rightarrow I'_{1i}$ holds.

These four cases prove Invariant I_{1i}. The proof for I_{2i} follows from the facts:

$now = m \times \rho_j$ iff $inv_j \leq (m+1)$ and $inv_j - com_j = 1$ and $T_{\tau_j} = now + C_j$

and

$\lceil now/\rho_j \rceil \geq inv_j$ if $now = m \times \rho_j + t_0$, where $0 < t_0 < \rho_j$

To prove that I_{3i} is an invariant, note that $MaxTime(T_{\tau_i})$ requires:

$$now' \leq T_{\tau_i} \leq C_i + Comp(i, now) \tag{12}$$

For any allowed transition A, assume that $C_i + Comp(i, now) \leq R_i \wedge A$. Then by I_{2i} and the inequation (12):

$$\begin{aligned} Comp(i, now') + C_i &\leq \sum_{j=1}^{i-1} \lceil now'/\rho_j \rceil \times C_j + C_i && I_{2i} \text{ of the lemma} \\ &\leq \sum_{j=1}^{i-1} \lceil (C_i + Comp(i, now))/\rho_j \rceil + C_i && \text{inequation (12)} \\ &\leq \sum_{j=1}^{i-1} \lceil R_i/\rho_j \rceil \times C_j + C_i R_i && \text{Definition of } R_i \end{aligned}$$

The general cases for I_{1i}, I_{2i} and I_{3i} can be proved in the same way. From the assumption that $R_i \leq D_i$, these three cases together guarantee that $\Box(T_{\tau_i} \leq H_{\tau_i})$ and thus the deadline for the task is always met. This ensures I_{4i} holds. Notice that I_{4i} is not used in the proof, though I_{4j}, for $j = 1, \ldots, i-1$, are used as the induction assumption. Therefore, we have proved the Lemma.

The proof of Theorem 1 follows Rule R3 in Section 7.4 in a straightforward way from this Lemma.

7.13 Discussion

The example in Section 7.9 deals with independent periodic tasks with fixed priorities. The method in scheduling theory used for these tasks has been extended to deal with communicating tasks. For example, tasks may communicate with each other asynchronously through a *protected shared object* (PSO) [40]. These tasks may be periodic or *sporadic*. For a scheduler with *ceiling priorities*, the worst response time R_i for a task τ_i can be calculated by the recurrence relation:

$$R_i^{(k+1)} = B_i + C_i + \sum_{j=1}^{i-1} \lceil \frac{R_i^{(k)}}{\rho_j} \rceil \times C_j$$

where B_i is the worst blocking time for τ_i by a task of lower priority, and ρ_j is minimum inter-arrival time of task τ_j (which is the period of τ_j if τ_j is periodic).

In the feasibility analysis of fault-tolerant real-time tasks [38], the recurrence relation for the worst response time R_i for a task τ_i has been extended to deal with fault-tolerant tasks: by re-execution of the affected task, by forward recovery, by recovery blocks, by checkpointing and backward recovery. In the case of fault-tolerance by re-execution, the response time R_i for a task τ_i can be calculated by the recurrence relation:

$$R_i^{(k+1)} = B_i + C_i + \sum_{j=1}^{i-1} \lceil \frac{R_i^{(k)}}{\rho_j} \rceil \times C_j + \lceil \frac{R_i^{(k)}}{F_j} \rceil \times max\{\ C_j\ \ 1 \leq j \leq i\ \}$$

where F_j is the minimum time between two occurrences of faults.

The formal method for scheduling analysis presented here can be applied to communicating, fault-tolerant tasks. This allows us to combine this work with our previous work on fault-tolerance and real-time [159, 162, 163, 164, 165], which formally treat re-execution, forward recovery, recovery blocks, and checkpointing and backward recovery, and provide a means of formally dealing with real-time program refinement, fault-tolerance and schedulability in a single and consistent framework.

Exercise 8. A Project for Self Study: Apply the notation and techniques to the development of the realtime mine pump system described in Section 1 (please see [39]). Then extend the solution to deal with fault-tolerance (please see Chapter 8 in [39]).

8 Related Work

There have been a number of other approaches to formalising real-time scheduling. Using the Duration Calculus [262], Zhou Chaochen et al [261, 260] have also separately specified a scheduler and a scheduled program. However, the Duration Calculus does not at present have powerful verification tools for proving program

refinement. It would be useful to unify the theories of Linear Temporal Logics and Duration Calculus for the specification and analysis of real-time systems. Work in this direction is making slow progress [170, 55, 56].

The work in [173] describes a case study using a *scheduling-oriented model for real-time systems* called TAM. The work of [91] extends Back's action systems [15] with timing and priorities and uses the **Z** notation for specification and refinement. The models used there appear to be more complicated than is necessary. For example, priorities and scheduling can be defined using only simple state variables and standard actions, as we have shown here, and complex models and structures are not needed.

Using timed CCS, [126, 127] deals with dynamic scheduling in the presence of faults by modelling resources and schedulers as processes. This serves well as a model but event-based process algebras tend to have a very different syntax to most traditional programming languages; it is possible to consider extensions to this work which make use of persistent timers and this would enable pre-emption to be modelled. The similar approach should also be applicable to the framework of CSP presented in Chapter 3 of this volume. For example, [204] deals with physical faults and verification of fault-tolerance by using the notation of CSP. As in our earlier work [169], these approaches use volatile time bounds (either explicitly or implicitly) for both program verification and scheduling analysis. When dealing with pre-emption (interruption) in real-time scheduling, the use of volatile time bounds requires a scheduled action to be *explicitly* divided into smaller actions (or steps) between whose execution pre-emption can occur. The atomicity of the original action has to be preserved and this requires the introduction of auxiliary internal variables. The feasibility of the implementation is established by reasoning about this *step-level* program. The use of these devices makes it difficult to reason about and make formal use of the methods and results from scheduling theory, especially as this does in fact make (informal) use of the accumulated execution time of tasks.

Another approach to the verification of schedulability uses algorithms for computing *quantitative* information of an implementation, such as the lower bound and upper bound on the delay between two (or two sets of) states [44, 43]. The quantitative information is then used to determine the feasibility of the implementation and to verify other timing properties using *symbolic model checking* techniques [184]. There are some significant differences between that work and what we have described:

1. The algorithms and the model-checking procedures described in [44] work effectively with a discrete time domain and a finite-state system; in contrast, in our analysis time is modelled by the reals and systems may have a finite or an infinite set of states.
2. Our framework allows program development through refinement to be integrated with scheduling theory so that the methods and results from the latter can be formal interpreted, verified and used correctly. [44] uses a scheduling algorithm to obtain an implementation and then tests for schedulability. There is no verification of whether a theorem in scheduling theory is valid for

the program model used (compare this with Section 7.9). In fact, application of Theorem 1 and the recurrence relation 8 to the Aircraft Control System example of [44, 43] leads directly to the same feasibility conclusion obtained there.

3. Compared with the work in [44, 43] which concentrates on timing aspects, this treatment deals with the much wider range of inter-related issues of concurrency, timing, fault-tolerance and schedulability, as well as refinement techniques for fault-tolerant and real-time programs. We also show how fault-tolerance affects schedulability in real-time applications.

In general, model checking techniques are especially effective and necessary in many safety-critical applications (please see Chapter 8 of this book on Model Checking). However, their general applicability has been restricted by questions of undecidability [9] and by complexity issues [7], especially for systems using a continuous time domain. These problems are very much more serious when both fault-tolerance and real-time have to be considered. Model-checking and the more general verification methods used here are complementary and neither can be totally replaced by the other. We recognize a role for model-checking as a decision procedure in a proof-checker, to be applied when possible.

It is usually impossible to give an exact prediction for the occurrence of faults in a program execution, or to achieve one hundred per cent fault-tolerance. Therefore, fault-tolerance is often addressed with the concepts of dependability and reliability. The occurrence of faults is associated with a probability distribution and verification of fault-tolerance is thus related to the calculation of reliability based on the probability distribution. There is no much work on formal models to support effective reasoning about reliability. We believe interesting work can be done by combining the model in this chapter with that of Chapter 4 on Probability. The idea of Unifying Theories of Programming in [117] will be very useful for this combination.

9 Conclusions

Formal development and verification of a real-time program requires a logical structure in which functional and timing properties of the program can be specified and reasoned about. In many practical cases, such programs are executed under a scheduler whose actions control the program's execution and thus its timing properties. A program is also often executed on a failure-prone system and thus fault-tolerance is needed. However, fault-tolerance and schedulability affect each other and they both affect the functionality and timing of the program. This chapter presents a framework which we believe is suitable for a coherent understanding of the relationship between theories of concurrency, real-time, fault-tolerance and schedulablity analysis; and for formal and systematic development of safety and/or timing critical computer systems.

Scheduling theory provides powerful techniques for determining the timing properties of a restricted class of real-time programs; however, it does not provide any means of verifying functional properties. Such methods must be augmented

by more traditional program verification techniques, but these use a different analytical framework, making it hard to relate the results in a rigorous way. This is particularly important when mechanized verification is to be performed and the program's properties certified, as is necessary in many safety-critical applications.

In a separate paper [169], we showed how the schedulability of a real-time program could be established using techniques very similar to those used here. An important observation that can be made about that work is that to simplify verification it is useful to reduce the number of actions by specifying them at *as high* a level as possible. However, for accurate verification of timing properties it is necessary to have a fine level of granularity in the time bounds for each action and each deadline: this requires specifying actions at *as low* a level as possible, so that pre-emption can be precisely modelled and the timing properties related to those obtained from scheduling theory.

We address this issue in this chapter by providing two kinds of timers: volatile timers that record times for which actions are continuously enabled, and persistent timers that sum the duration for which actions are executed. The use of persistent timers allows the timing effects of lower-level actions, like pre-emption, to be considered abstractly and at a higher-level. It no longer matters exactly when an action is pre-empted: what is important is the time for which it executed before pre-emption and the time for which it is pre-empted. Thus an action may be pre-empted a number of times and still make use of a single timer to record its timing properties.

The use of two kinds of timers solves a problem that has been the cause of a major restriction in the application of formal verification methods in the validation of real-time programs. It makes it feasible to use automated verification for such programs at the specification level, allowing timing properties to be considered well before the details of the implementation have been finalised. Naturally, once the implementation is complete, scheduling analysis will still be required to validate and provide independent certification of the timing properties.

The method presented in this chapter is independent of a programming language. Also, both the program and the scheduler specifications can be refined, with feasibility and correctness being preserved at each step. This has the great advantage that proving feasibility does not first require the code of the program to be developed.

There are many advantages to using a single, consistent treatment of fault-tolerance, timing and schedulability. Not only does it allow a unified view to be taken of the functional and non-functional properties of programs and a simple transformational method to be used to combine these properties, it also makes it possible to use a uniform method of verification. Verification of schedulability within a proof framework will inevitably be more cumbersome than using a simple schedulability test from scheduling theory. However, the use of a common framework means that during formal verification, the test for schedulability can be defined as a theorem whose verification is not actually done within the

proof theory but instead by invoking an oracle or decision procedure which uses scheduling theory for rapid analysis.

The plan of our future work includes the combination of the techniques presented in this chapter with those developed in our recent work on object-oriented and component based systems [129, 160]. We hope such a combination will lead to a multi-view and multi-notational framework for modelling, design, analysis and verification of real-time and fault-tolerant systems at different levels of abstraction. It will also support transformational, incremental and iterative development [161, 258] aided with transformation and verification tools [154, 171, 241].

A Tutorial Introduction to CSP in *Unifying Theories of Programming*

Ana Cavalcanti and Jim Woodcock

Department of Computer Science
University of York
York, UK

In their *Unifying Theories of Programming* (UTP), Hoare & He use the alphabetised relational calculus to give denotational semantics to a wide variety of constructs taken from different programming paradigms. In this chapter, we give a tutorial introduction to the semantics of CSP processes, as presented in Chapter 3. We start with a summarised introduction of the alphabetised relational calculus and the theory of designs, which are pre-post specifications in the style of specification statements. Afterwards, we present in detail a theory for reactive processes. Later, we combine the theories of designs and reactive processes to provide the model for CSP processes. Finally, we compare this new model with the standard failures-divergences model for CSP.

In the next section, we give an overview of the UTP, and in Section 2 we present its most general theory: the alphabetised predicates. In the following section, we establish that this theory is a complete lattice. Section 4 restricts the general theory to designs. Section 5 presents the theory of reactive processes; Section 6 contains our treatment of CSP processes; and Section 7 relates our model to Roscoe's standard model. We summarise the work in Section 8.

1 Introduction

The book by Hoare & He [117] sets out a research programme to find a common basis in which to explain a wide variety of programming paradigms: unifying theories of programming (UTP). Their technique is to isolate important language features, and give them a denotational semantics. This allows different languages and paradigms to be compared.

The semantic model is an alphabetised version of Tarski's relational calculus, presented in a predicative style that is reminiscent of the schema calculus in the Z [257] notation. Each programming construct is formalised as a relation between an initial and an intermediate or final observation. The collection of these relations forms a *theory* of the paradigm being studied, and it contains three essential parts: an alphabet, a signature, and healthiness conditions.

The *alphabet* is a set of variable names that gives the vocabulary for the theory being studied. Names are chosen for any relevant external observations of behaviour. For instance, programming variables x, y, and z would be part of the alphabet. Also, theories for particular programming paradigms require the observation of extra information; some examples are a flag that says whether

A. Cavalcanti, A. Sampaio, and J. Woodcock (Eds.): PSSE 2004, LNCS 3167, pp. 220–268, 2006.

the program has started (*okay*); the current time (*clock*); the number of available resources (*res*); a trace of the events in the life of the program (*tr*); or a flag that says whether the program is waiting for interaction with its environment (*wait*). The *signature* gives the rules for the syntax for denoting objects of the theory. *Healthiness conditions* identify properties that characterise the theory.

Each healthiness condition embodies an important fact about the computational model for the programs being studied.

Example 1 (Healthiness conditions).

1. The variable *clock* gives us an observation of the current time, which moves ever onwards. The predicate B specifies this.

   $$B \mathrel{\widehat{=}} clock \leq clock'$$

   If we add B to the description of some activity, then the variable *clock* describes the time observed immediately before the activity starts, whereas $clock'$ describes the time observed immediately after the activity ends. If we suppose that P is a healthy program, then we must have that $P \Rightarrow B$.
2. The variable *okay* is used to record whether or not a program has started. A sensible healthiness condition is that we should not observe a program's behaviour until it has started; such programs satisfy the following equation.

   $$P = (okay \Rightarrow P)$$

   If the program has not started, its behaviour is not restricted.

Healthiness conditions can often be expressed in terms of a function ϕ that makes a program healthy. There is no point in applying ϕ twice, since we cannot make a healthy program even healthier. Therefore, ϕ must be idempotent, and a healthy P must be a fixed point: $P = \phi(P)$; this equation characterises the healthiness condition. For example, we can turn the first healthiness condition above into an equivalent equation, $P = P \wedge B$, and then the following function on predicates $and_B \mathrel{\widehat{=}} \lambda X \bullet X \wedge B$ is the required idempotent.

The relations are used as a semantic model for unified languages of specification and programming. Specifications are distinguished from programs only by the fact that the latter use a restricted signature. As a consequence of this restriction, programs satisfy a richer set of healthiness conditions.

Unconstrained relations are too general to handle the issue of program termination; they need to be restricted by healthiness conditions. The result is the theory of designs, which is the basis for the study of the other programming paradigms in [117]. Here, we present the general relational setting, and the transition to the theory of designs. Next we take a different tack, and introduce the theory of reactive processes, which we then combine with designs to form the theory of CSP [115, 225].

2 The Alphabetised Relational Calculus

The alphabetised relational calculus is similar to Z's schema calculus, except that it is untyped and rather simpler. An *alphabetised predicate* $(P, Q, \ldots, \mathbf{true})$ is an alphabet-predicate pair, where the predicate's free variables are all members of the alphabet. Relations are predicates in which the alphabet is composed of undecorated variables $(x,\ y,\ z,\ \ldots)$ and dashed variables $(x',\ a',\ \ldots)$; the former represent initial observations, and the latter, observations made at a later intermediate or final point.

The alphabet of an alphabetised predicate P is denoted αP, and may be divided into its before-variables ($in\alpha P$) and its after-variables ($out\alpha P$). A *homogeneous relation* has $out\alpha P = in\alpha P'$, where $in\alpha P'$ is the set of variables obtained by dashing all variables in the alphabet $in\alpha P$. A *condition* $(b, c, d, \ldots, true)$ has an empty output alphabet.

Standard predicate calculus operators can be used to combine alphabetised predicates. Their definitions, however, have to specify the alphabet of the combined predicate. For instance, the alphabet of a conjunction is the union of the alphabets of its components: $\alpha(P \wedge Q) = \alpha P \cup \alpha Q$. If a variable is mentioned in the alphabet of P and Q, then they are both constraining the same variable.

A distinguishing feature of the UTP is its concern with program development, and consequently program correctness. A significant achievement is that the notion of program correctness is the same in every paradigm in [117]: in every state, the behaviour of an implementation implies its specification.

If we suppose that $\alpha P = \{a, b, a', b'\}$, then the *universal closure* of P is simply $\forall\, a, b, a', b' \bullet P$, which is more concisely denoted as $[\,P\,]$. The correctness of a program P with respect to a specification S is denoted by $S \sqsubseteq P$ (S is refined by P), and is defined as follows.

$$S \sqsubseteq P \quad \text{iff} \quad [\,P \Rightarrow S\,]$$

Example 2 (Refinement). Suppose we have the specification $x' > x \wedge y' = y$, and the implementation $x' = x + 1 \wedge y' = y$. The implementation's correctness can be argued as follows.

$$\begin{array}{lll} & x' > x \wedge y' = y \ \sqsubseteq\ x' = x + 1 \wedge y' = y & \sqsubseteq \\ = & [\,x' = x + 1 \wedge y' = y \Rightarrow x' > x \wedge y' = y\,] & \text{universal one-point rule, twice} \\ = & [\,x + 1 > x \wedge y = y\,] & \text{arithmetic and reflection} \\ = & true & \end{array}$$

And so, the refinement is valid.

As a first example of the definition of a programming constructor, we consider conditionals. Hoare & He use an infix syntax for the conditional operator, and define it as follows.

$$P \lhd b \rhd Q \;\widehat{=}\; (b \wedge P) \vee (\neg\, b \wedge Q) \qquad \textbf{if } \alpha b \subseteq \alpha P = \alpha Q$$
$$\alpha(P \lhd b \rhd Q) \;\widehat{=}\; \alpha P$$

Informally, $P \lhd b \rhd Q$ means P if b else Q.

The presentation of conditional as an infix operator allows the formulation of many laws in a helpful way. Below, we reproduce some of the laws presented in [117].

L1	$P \lhd b \rhd P = P$	*idempotence*
L2	$P \lhd b \rhd Q = Q \lhd \neg\, b \rhd P$	*symmetry*
L3	$(P \lhd b \rhd Q) \lhd c \rhd R = P \lhd b \wedge c \rhd (Q \lhd c \rhd R)$	*associativity*
L4	$P \lhd b \rhd (Q \lhd c \rhd R) = (P \lhd b \rhd Q) \lhd c \rhd (P \lhd b \rhd R)$	*distributivity*
L5	$P \lhd \mathit{true} \rhd Q = P = Q \lhd \mathit{false} \rhd P$	*unit*
L6	$P \lhd b \rhd (Q \lhd b \rhd R) = P \lhd b \rhd R$	*unreachable-branch*
L7	$P \lhd b \rhd (P \lhd c \rhd Q) = P \lhd b \vee c \rhd Q$	*disjunction*
L8	$(P \odot Q) \lhd b \rhd (R \odot S) = (P \lhd b \rhd R) \odot (Q \lhd b \rhd S)$	*interchange*

In Law **L8**, the symbol $\odot$ stands for any truth-functional operator.

For each operator, Hoare & He give a definition followed by a number of algebraic laws as those above. These laws can be proved from the definition; proofs omitted here can be found in [117] or [256]. We also present extra laws that are useful in later proofs, as well as in illuminating the theory. We give the laws presented in [117] that we reproduce here the same labels used in that original work: **L1**, **L2** and so on. The extra laws that we present are numbered independently.

Since a conditional is just an abbreviation for a predicate, for reasoning, we can use laws that combine programming and predicate calculus operators. An example is our first law below, which states that negating a conditional negates its operands, but not its condition.

Law 60 (not-conditional). $\neg\,(P \lhd b \rhd Q) = (\neg\, P \lhd b \rhd \neg\, Q)$

Proof.

$$\begin{aligned}
&\neg\,(P \lhd b \rhd Q) && \text{conditional}\\
&= \neg\,((b \wedge P) \vee (\neg\, b \wedge Q)) && \text{propositional calculus}\\
&= (b \Rightarrow \neg\, P) \wedge (\neg\, b \Rightarrow \neg\, Q) && \text{propositional calculus}\\
&= (b \wedge \neg\, P) \vee (\neg\, b \wedge \neg\, Q) && \text{conditional}\\
&= (\neg\, P \lhd b \rhd \neg\, Q)
\end{aligned}$$

If we apply the law of symmetry to the last result, we see that negating a conditional can be used to negate its condition, but in this case, the operands must be both negated and reversed: $\neg\,(P \lhd b \rhd Q) = (\neg\, Q \lhd \neg\, b \rhd \neg\, P)$. Even though it does not make sense to use negation in a program, for reasoning, the flexibility is very convenient.

Below is an instance of Law **L8** with a compound truth-functional operator.

Law 61 (conditional-and-not-conditional).

$$(P \lhd b \rhd Q) \wedge \neg (R \lhd b \rhd S) = (P \wedge \neg R) \lhd b \rhd (Q \wedge \neg S)$$

Proof.

$$\begin{array}{ll} & (P \lhd b \rhd Q) \wedge \neg (R \lhd b \rhd S) & \text{Law 60} \\ = & (P \lhd b \rhd Q) \wedge (\neg R \lhd b \rhd \neg S) & \text{L8} \\ = & (P \wedge \neg R) \lhd b \rhd (Q \wedge \neg S)) & \end{array}$$

As a consequence of the interchange (**L8**) and unit (**L1**) laws, any boolean operator distributes through the conditional.

Law 62 ($\odot$-conditional).

$$(P \odot (Q \lhd b \rhd R)) = ((P \odot Q) \lhd b \rhd (P \odot R))$$
$$((P \lhd b \rhd Q) \odot R) = ((P \odot R) \lhd b \rhd (Q \odot R))$$

The details of this simple proof and of others omitted in the sequel are left as exercises. We include here only proofs for the more surprising laws or proofs that perhaps require more elaborate arguments.

Exercise 1. Prove Law 62.

A conditional may be simplified by using a known condition.

Law 63 (known-condition).

$$b \wedge (P \lhd b \rhd Q) = (b \wedge P)$$
$$\neg\, b \wedge (P \lhd b \rhd Q) = (\neg\, b \wedge Q)$$

Two absorption laws allow a conditional's operands to be simplified.

Law 64 (assume-if-condition). $(P \lhd b \rhd Q) = ((b \wedge P) \lhd b \rhd Q)$

Law 65 (assume-else-condition). $(P \lhd b \rhd Q) = (P \lhd b \rhd (\neg\, b \wedge Q))$

Sequence is modelled as relational composition. Two relations may be composed providing the output alphabet of the first is the same as the input alphabet of the second, except only for the use of dashes.

$$P(v') \,;\, Q(v) \;\widehat{=}\; \exists\, v_0 \bullet P(v_0) \wedge Q(v_0) \qquad \textbf{if } out\alpha P = in\alpha Q' = \{v'\}$$
$$in\alpha(P(v') \,;\, Q(v)) \;\widehat{=}\; in\alpha P$$
$$out\alpha(P(v') \,;\, Q(v)) \;\widehat{=}\; out\alpha Q$$

Sequence is associative and distributes backwards through the conditional.

$$\textbf{L1} \quad P \,;\, (Q \,;\, R) = (P \,;\, Q) \,;\, R \qquad \textit{associativity}$$
$$\textbf{L2} \quad (P \lhd b \rhd Q) \,;\, R = ((P \,;\, R) \lhd b \rhd (Q \,;\, R)) \qquad \textit{left-distribution}$$

The definition of assignment is basically equality; we need, however, to be careful about the alphabet. If $A = \{x, y, \ldots, z\}$ and $\alpha e \subseteq A$, where αe is the set of free variables of the expression e, the assignment $x :=_A e$ of expression e to variable x changes only x's value.

$$x :=_A e \ \widehat{=}\ (x' = e \wedge y' = y \wedge \cdots \wedge z' = z)$$
$$\alpha(x :=_A e) \ \widehat{=}\ A \cup A'$$

There is a degenerate form of assignment that changes no variable: it has the following definition.

$$\mathit{II}_A \ \widehat{=}\ (v' = v) \qquad \textbf{if } A = \{v\}$$
$$\alpha \mathit{II}_A \ \widehat{=}\ A \cup A'$$

Here, v stands for a list of observational variables. We use $v' = v$ to denote the conjunction of equalities $x' = x$, for all x in v. When clear from the context, we omit the alphabet of assignments and II.

II is the identity of sequence.

L5 $\quad P \,;\, \mathit{II}_{\alpha P} = P = \mathit{II}_{\alpha P} \,;\, P \qquad$ *unit*

Since sequence is defined in terms of the existential quantifier, there are two one-point laws. We prove one of them; the proof of the other is a simple exercise.

Law 66 (left-one-point). $(v' = e) \,;\, P \ = \ P[e/v]$
provided $\alpha P = \{v, v'\}$ *and* v' *is not free in* e.

Law 67 (right-one-point). $P \,;\, (v = e) \ = \ P[e/v']$
provided $\alpha P = \{v, v'\}$ *and* v *is not free in* e.

Proof.

$$\begin{array}{lll}
 & P \,;\, v = \mathbf{e} & \text{sequence} \\
= & \exists\, v_0 \bullet P[v_0/v'] \wedge (v = \mathbf{e})[v_0/v] & \text{substitution} \\
= & \exists\, v_0 \bullet P[v_0/v'] \wedge (v_0 = \mathbf{e}) & \text{predicate calculus and } v \text{ not free in } e \\
= & P[v_0/v'][\mathbf{e}/v_0] & \text{substitution} \\
= & P[\mathbf{e}/v'] &
\end{array}$$

Exercise 2. Prove Law **L7** above.

In theories of programming, nondeterminism may arise in one of two ways: either as the result of run-time factors, such as distributed processing; or as the under-specification of implementation choices. Either way, nondeterminism is modelled by choice; the semantics is simply disjunction.

$$P \sqcap Q \ \widehat{=}\ P \vee Q \qquad \textbf{if } \alpha P = \alpha Q$$
$$\alpha(P \sqcap Q) \ \widehat{=}\ \alpha P$$

The alphabet must be the same for both arguments.

Variable blocks are split into the commands $\mathbf{var}\, x$, which declares and introduces x in scope, and $\mathbf{end}\, x$, which removes x from scope. Their definitions are presented below, where A is an alphabet containing x and x'.

$$\mathbf{var}\, x \mathrel{\widehat=} (\exists x \bullet \Pi_A) \qquad \alpha(\mathbf{var}\, x) \mathrel{\widehat=} A \setminus \{x\}$$

$$\mathbf{end}\, x \mathrel{\widehat=} (\exists x' \bullet \Pi_A) \qquad \alpha(\mathbf{end}\, x) \mathrel{\widehat=} A \setminus \{x'\}$$

The relation $\mathbf{var}\, x$ is not homogeneous, since it does not include x in its alphabet, but it does include x'; similarly, $\mathbf{end}\, x$ includes x, but not x'.

The results below state that following a variable declaration by a program Q makes x local in Q; similarly, preceding a variable undeclaration by a program Q makes x' local.

$$(\mathbf{var}\, x ; Q) = (\exists x \bullet Q)$$

$$(Q ; \mathbf{end}\, x) = (\exists x' \bullet Q)$$

More interestingly, we can use $\mathbf{var}\, x$ and $\mathbf{end}\, x$ to specify a variable block.

$$(\mathbf{var}\, x ; Q ; \mathbf{end}\, x) = (\exists x, x' \bullet Q)$$

In programs, we use $\mathbf{var}\, x$ and $\mathbf{end}\, x$ paired in this way, but the separation is useful for reasoning.

Exercise 3. Prove the above equality.

Variable blocks introduce the possibility of writing programs and equations like that below.

$$(\mathbf{var}\, x ; x := 2 * y ; w := 0 ; \mathbf{end}\, x) \\ = (\mathbf{var}\, x ; x := 2 * y ; \mathbf{end}\, x) ; w := 0$$

Clearly, the assignment to w may be moved out of the scope of the declaration of x, but what is the alphabet in each of the assignments to w? If the only variables are w, x, and y, and $A = \{w, y, w', y'\}$, then the assignment on the right has the alphabet A; but the alphabet of the assignment on the left must also contain x and x', since they are in scope. There is an explicit operator for making alphabet modifications such as this: *alphabet extension*.

$$P_{+x} \mathrel{\widehat=} P \wedge x' = x \qquad \textbf{for}\ \ x, x' \notin \alpha P$$
$$\alpha(P_{+x}) \mathrel{\widehat=} \alpha P \cup \{x, x'\}$$

In our example, if the right-hand assignment is $P \mathrel{\widehat=} w :=_A 0$, then the left-hand assignment is denoted by P_{+x}.

The next programming operator of interest is recursion. We define it in the next section, where we explain that the UTP general theory of relations is a complete lattice, a notion introduced in Chapter 0.

3 The Complete Lattice

As already explained in Chapter 0, the refinement ordering is a partial order: reflexive, anti-symmetric, and transitive; this also holds for refinement as defined in the UTP. Moreover, the set of alphabetised predicates with a particular alphabet A is a complete lattice under the refinement ordering. Its bottom element is denoted $\bot_{\mathrm{A}}$, and is the weakest predicate **true**; this is the program that behaves quite arbitrarily. The top element is denoted $\top_A$, and is the strongest predicate ***false***; this is the program that performs miracles and implements every specification (see Chapter 0). These properties of abort and miracle are captured in the following two laws, which hold for all P with alphabet A.

L1 $\bot_{\mathrm{A}} \sqsubseteq P$ *bottom-element*

L2 $P \sqsubseteq \top_{\mathrm{A}}$ *top-element*

The least upper bound is not defined in terms of the relational model, but by the Law **L1** below; this is because, in general, it is not possible to give such definition. Fortunately, this law indirectly specifies the least upper bound operator; alone, it is enough to prove Laws **L1A** and **L1B**, which are actually more useful in proofs.

L1 $P \sqsubseteq (\sqcap S)$ iff $(P \sqsubseteq X$ for all X in $S)$ *unbounded-nondeterminism*

L1A $(\sqcap S) \sqsubseteq X$ for all X in S *lower-bound*

L1B if $P \sqsubseteq X$ for all X in S, then $P \sqsubseteq (\sqcap S)$ *greatest-lower-bound*

These laws characterise basic properties of least upper bounds. In particular, Law **L1B** is simply **L1**, from right to left.

A function F is *monotonic* if, and only if, $P \sqsubseteq Q \Rightarrow F(P) \sqsubseteq F(Q)$. Operators like conditional and sequence are monotonic; negation is not. There is a class of operators that are all monotonic: the disjunctive operators. For example, sequence is disjunctive in both arguments.

L6 $(P \sqcap Q)\,;\, R = (P\,;\, R) \sqcap (Q\,;\, R)$ *sequence-⊓-left-distribution*

L7 $P\,;\,(Q \sqcap R) = (P\,;\, Q) \sqcap (P\,;\, R)$ *sequence-⊓-right-distribution*

Exercise 4. Prove Law **L6** above.

Since alphabetised relations form a complete lattice, every construction defined solely using monotonic operators has a fixed point (see Chapter 1, Section 8). Even more, a result by Tarski says that the set of fixed points is a complete

lattice. The extreme points in this lattice are often of interest; for example, $\top$ is the strongest fixed point of $X = X \,;\, P$, and $\bot$ is the weakest.

The weakest fixed point of the function F is denoted by $\mu\, F$, and is simply the greatest lower bound (the *weakest*) of all the fixed points of F.

$$\mu\, F \;\widehat{=}\; \sqcap\{\, X \mid F(X) \sqsubseteq X \,\}$$

The strongest fixed point νF is the dual of the weakest fixed point.

Hoare & He use weakest fixed points to define recursion. They write a recursive program as $\mu\, X \bullet \mathcal{C}(X)$, where $\mathcal{C}(X)$ is a predicate that is constructed using monotonic operators and the variable X. As opposed to the variables in the alphabet, X stands for a predicate itself, and we call it the recursive variable. Intuitively, occurrences of X in $\mathcal{C}$ stand for recursive calls to $\mathcal{C}$ itself. The definition of recursion is as follows.

$$\mu\, X \bullet \mathcal{C}(X) \;\widehat{=}\; \mu\, F \qquad \textbf{where } F \;\widehat{=}\; \lambda\, X \bullet \mathcal{C}(X)$$

The standard laws that characterise weakest fixed points are valid.

L1 $\quad \mu\, F \sqsubseteq Y$ if $F(Y) \sqsubseteq Y$ *weakest-fixed-point*

L2 $\quad F(\mu\, F) = \mu\, F$ *fixed-point*

Law **L1** establishes that $\mu\, F$ is weaker than any fixed point; **L2** states that $\mu\, F$ is itself a fixed point. From a programming point of view, **L2** is just the copy rule.

The while loop is written $b * P$: while b is true, execute the program P. This can be defined in terms of the weakest fixed point of a conditional expression.

$$b * P \;\widehat{=}\; \mu\, X \bullet (\,(P \,;\, X) \lhd b \rhd \mathit{II}\,)$$

Example 3 (Non-termination). If b always remains true, then obviously the loop $b * P$ never terminates, but what is the semantics for this? The simplest example of such an iteration is $\mathit{true} * \mathit{II}$, which has the semantics $\mu\, X \bullet X$.

$$\begin{array}{lr}
\quad \mu\, X \bullet X & \text{least fixed point} \\
= \sqcap\{\, Y \mid (\lambda\, X \bullet X)(Y) \sqsubseteq Y \,\} & \text{function application} \\
= \sqcap\{\, Y \mid Y \sqsubseteq Y \,\} & \text{reflexivity of } \sqsubseteq \\
= \sqcap\{\, Y \mid \mathit{true} \,\} & \text{property of } \sqcap \\
= \bot &
\end{array}$$

Exercise 5. Convince yourself that $\mathit{true} * \mathit{II} = \mu\, X \bullet X$ using the laws presented so far.

Surprisingly, it is possible to use the result Example 3 to show that a program may be able to recover from a non-terminating loop!

Example 4 (Aborting loop). Suppose that the sole state variable is x and that c is a constant.

$$
\begin{array}{lr}
(\mathbf{true} * II)\ ;\ x := c & \text{Example 3} \\
= \bot\ ;\ x := c & \bot \\
= \mathbf{true}\ ;\ x := c & \text{assignment} \\
= \mathbf{true}\ ;\ x' = c & \text{sequence} \\
= \exists\, x_0 \bullet \mathbf{true} \wedge x' = c & \text{predicate calculus} \\
= x' = c & \text{assignment} \\
= x := c &
\end{array}
$$

Example 4 is rather disconcerting: in ordinary programming, there is no recovery from a non-terminating loop. It is the purpose of *designs* to overcome this deficiency in the programming model.

4 Designs

The problem pointed out above in Section 3 can be explained as the failure of general alphabetised predicates P to satisfy the equation below.

$$\mathbf{true}\ ;\ P \ =\ \mathbf{true}$$

We presented in Example 4 a program consisting of a non-terminating loop followed by an assignment, and whose overall behaviour was to ignore the loop and execute the assignment. This is not how programs work in practice. The solution to this problem is to consider a subset of the alphabetised predicates in which a particular observational variable, called *okay*, is used to record information about the start and termination of programs. The above equation holds for predicates P in this set. As an aside, note that **false** cannot possibly belong to this set, since $\mathbf{true}\ ;\ \mathbf{false} = \mathbf{false}$.

The predicates in this subset are called *designs*. They can be split into precondition-postcondition pairs, and are a basis for unifying languages and methods like B [3], VDM [134], Z [257], and refinement calculi [192, 17, 199]. They are similar to the specification statements introduced in Chapter 0.

In designs, *okay* records that the program has started, and $okay'$ that it has terminated. In implementing a design, we may assume that the precondition holds, but we have to fulfill the postcondition. In addition, we can rely on the program being started, but we must ensure that it terminates. If the precondition does not hold, or the program does not start, we are not committed to establish the postcondition nor even to make the program terminate.

A design with precondition P and postcondition Q is written $(\, P \vdash Q \,)$. It is defined as follows.

$$(\, P \vdash Q \,) \ \widehat{=}\ (\, okay \wedge P \Rightarrow okay' \wedge Q \,)$$

If the program starts in a state satisfying P, then it will terminate, and on termination Q will be true.

Example 5 (Pre-post specifications). Suppose that we have a program with state variables x and y, and we want to specify that, providing that x is strictly positive, x must be decreased whilst y is kept constant. The design that formalises this is

$$x > 0 \vdash x' < x \land y' = y$$

This is more or less the same in Morgan's refinement calculus, where the specification statement below could be used.

$$x : [\, x > 0, x < x_0 \,]$$

Notice how the postcondition uses different conventions for distinguishing between before and after variables; also notice that the specification is prefixed with a *frame* listing the variables that are permitted to change in order to satisfy the postcondition.

Abort and miracle are defined as designs in the following examples. Abort has precondition **false**: it is never guaranteed to terminate.

Example 6 (Abort).

$$\begin{array}{lr}
\quad \mathbf{false} \vdash \mathbf{false} & \text{design} \\
= \mathit{okay} \land \mathbf{false} \Rightarrow \mathit{okay}' \land \mathbf{false} & \mathbf{false} \text{ zero for conjunction} \\
= \mathbf{false} \Rightarrow \mathit{okay}' \land \mathbf{false} & \text{vacuous implication} \\
= \mathbf{true} & \text{vacuous implication} \\
= \mathbf{false} \Rightarrow \mathit{okay}' \land \mathbf{true} & \mathbf{false} \text{ zero for conjunction} \\
= \mathit{okay} \land \mathbf{false} \Rightarrow \mathit{okay}' \land \mathbf{true} & \text{design} \\
= \mathbf{false} \vdash \mathbf{true} &
\end{array}$$

Miracle has precondition **true**, and establishes the impossible: **false**.

Example 7 (Miracle).

$$\begin{array}{lr}
\quad \mathbf{true} \vdash \mathbf{false} & \text{design} \\
= \mathit{okay} \land \mathbf{true} \Rightarrow \mathit{okay}' \land \mathbf{false} & \mathbf{true} \text{ unit for conjunction} \\
= \mathit{okay} \Rightarrow \mathbf{false} & \text{contradiction} \\
= \neg\, \mathit{okay} &
\end{array}$$

Exercise 6. Prove that abort is refined by every other design, and that every design is refined by miracle.

In VDM, B, and the refinement calculus, a pre-post specification may be refined by weakening the precondition. This refinement step improves the specification, since there are some states in which execution of the original specification leads to abortion, but execution of the resulting specification has a well-defined behaviour.

A pre-post specification may also be refined by strengthening the postcondition. Again, this is an improvement, since more is known about the result.

Example 8 (Refining designs). In Example 5, the design aborts unless $x > 0$; we can improve this by requiring it to work when $x = 0$. This weakens the precondition, since $x > 0 \Rightarrow x \geq 0$. The design requires that the after-value of x should be strictly less than the before-value of x. We can strengthen this by saying how much smaller it should be. For instance, we could require that $x' = x - 1$. Moreover, we can weaken the precondition and strengthen the postcondition simultaneously. For example, the design below is a refinement of that in Example 5.

$$x \geq 0 \vdash (x > 0 \Rightarrow x' = 0) \wedge (x = 0 \Rightarrow x' = 1)$$

The behaviour for $x = 0$ is not related to that for when $x > 0$. This is an improvement in the sense that, within the old precondition, the new postcondition is stronger than the old postcondition.

We saw earlier that refinement between relations is just reverse implication; since designs are a special case of relations, it would be nice if the notion of refinement did not change. A reassuring result is that refinement of designs in the relational sense does amount to either weakening the precondition, or strengthening the postcondition in the presence of the precondition as expected. This is established by the result below.

Law 68 (refinement-of-designs).

$$P_1 \vdash Q_1 \sqsubseteq P_2 \vdash Q_2 = [\, P_1 \wedge Q_2 \Rightarrow Q_1 \,] \wedge [\, P_1 \Rightarrow P_2 \,]$$

Proof.

$$\begin{array}{lr}
& P_1 \vdash Q_1 \sqsubseteq P_2 \vdash Q_2 \qquad \sqsubseteq \\
= & [\,(\, P_2 \vdash Q_2 \,) \Rightarrow (\, P_1 \vdash Q_1 \,)\,] \qquad \text{definition of design, twice} \\
= & [\,(\, \mathit{okay} \wedge P_2 \Rightarrow \mathit{okay}' \wedge Q_2 \,) \Rightarrow (\, \mathit{okay} \wedge P_1 \Rightarrow \mathit{okay}' \wedge Q_1 \,)\,] \\
& \qquad \text{case split } \mathit{okay} \\
= & [\,(\, P_2 \Rightarrow \mathit{okay}' \wedge Q_2 \,) \Rightarrow (\, P_1 \Rightarrow \mathit{okay}' \wedge Q_1 \,)\,] \qquad \text{case split } \mathit{okay}' \\
= & [\,(\neg\, P_2 \Rightarrow \neg\, P_1 \,) \wedge ((\, P_2 \Rightarrow Q_2 \,) \Rightarrow (\, P_1 \Rightarrow Q_1 \,))\,] \\
& \qquad \text{propositional calculus} \\
= & [\,(\, P_1 \Rightarrow P_2 \,) \wedge ((\, P_2 \Rightarrow Q_2 \,) \Rightarrow (\, P_1 \Rightarrow Q_1 \,))\,] \qquad \text{predicate calculus} \\
= & [\, P_1 \Rightarrow P_2 \,] \wedge [\, P_1 \wedge Q_2 \Rightarrow Q_1 \,]
\end{array}$$

Exercise 7. Use Law 68 to prove the refinements in Example 8.

Sometimes, we need to refer to the precondition in the postcondition; this is called *exporting the precondition.*

Lemma 1 (export-precondition).

$$(P \vdash Q) = (P \vdash P \wedge Q)$$

Proofs of this lemma and of some our other lemmas and theorems can be found in Appendix B. They are usually results stated, but possibly not proved in [117], and results that we need in the proofs of our laws.

The most important result in the theory of designs, however, is that abort is a zero for sequence. This was, after all, the whole point for the introduction of designs. First, we introduce a lemma relating designs and abort.

Lemma 2 (design-abort). *When a design has not started ($\neg$ okay), it offers no guarantees.*

$$(P \vdash Q)[\mathit{false}/\mathit{okay}] = \mathbf{true}$$

This result holds for miracle as well, because even miracle cannot help if it does not start.

The left-zero law now follows from this lemma, since one possibility for abort is to make okay' false. If this is followed by a design, then the design will attempt to start in a state with *okay* false, and Lemma 2 will be relevant.

L1 $\quad \mathbf{true} \,;\, (P \vdash Q) = \mathbf{true}$ *left-zero*

Proof.

$$\begin{array}{ll}
& \mathbf{true} \,;\, (P \vdash Q) & \text{sequence} \\
= & \exists\, \mathit{okay}_0, v_0 \bullet \mathbf{true}[\mathit{okay}_0, v_0/\mathit{okay}', v'] \wedge (P \vdash Q)[\mathit{okay}_0, v_0/\mathit{okay}, v] & \\
& & \text{predicate calculus} \\
= & \exists\, \mathit{okay}_0 \bullet \exists\, v_0 \bullet \mathbf{true}[\mathit{okay}_0/\mathit{okay}'][v_0/v'] \wedge (P \vdash Q)[\mathit{okay}_0/\mathit{okay}][v_0/v] & \\
& & \text{sequence} \\
= & \exists\, \mathit{okay}_0 \bullet \mathbf{true}[\mathit{okay}_0/\mathit{okay}'] \,;\, (P \vdash Q)[\mathit{okay}_0/\mathit{okay}] & \text{case split } \mathit{okay}_0 \\
= & \mathbf{true}[\mathit{true}/\mathit{okay}'] \,;\, (P \vdash Q)[\mathit{true}/\mathit{okay}] \vee & \text{substitution, design-abort} \\
& \mathbf{true}[\mathit{false}/\mathit{okay}'] \,;\, (P \vdash Q)[\mathit{false}/\mathit{okay}] & \\
= & \mathbf{true} \,;\, (P \vdash Q)[\mathit{true}/\mathit{okay}] \vee \mathbf{true} \,;\, \mathbf{true} & \text{relational calculus} \\
= & \mathbf{true} &
\end{array}$$

In this new setting, it is necessary to redefine assignment and $\mathit{I\!I}$, as those introduced previously are not designs.

$$(x := e) \;\widehat{=}\; (\,\mathbf{true} \vdash x' = e \wedge y' = y \wedge \cdots \wedge z' = z\,)$$

$$\mathit{I\!I}_D \;\widehat{=}\; (\,\mathbf{true} \vdash \mathit{I\!I}\,)$$

Their existing laws hold, but it is necessary to prove them again, as their definitions have changed.

L2 $\quad (v := e \,;\, v := f(v)) = (v := f(e))$ *assignment-composition*

L3 $\quad (v := e \,;\, (P \lhd b(v) \rhd Q)) = ((v := e \,;\, P) \lhd b(e) \rhd (v := e \,;\, Q))$ *assignment-conditional-left-distribution*

L4 $\quad (\mathit{I\!I}_D \,;\, (P \vdash Q)) = (P \vdash Q)$ *left-unit*

Proof of **L2**.

$$
\begin{array}{ll}
& v := e \,;\, v := f(v) & \text{assignment, twice} \\
= & (\, \mathbf{true} \vdash v' = e \,) \,;\, (\, \mathbf{true} \vdash v' = f(v) \,) & \text{sequence,case split } okay_0 \\
= & (\, (\, \mathbf{true} \vdash v' = e \,)[true/okay'] \,;\, (\, \mathbf{true} \vdash v' = f(v) \,)[true/okay] \,) \vee \\
& \neg\, okay \,;\, \mathbf{true} & \text{design} \\
= & (\, (\, okay \Rightarrow v' = e \,) \,;\, (\, okay' \wedge v' = f(v) \,) \,) \vee \neg\, okay & \text{relational calculus} \\
= & okay \Rightarrow (\, v' = e \,;\, (\, okay' \wedge v' = f(v) \,) \,) & \text{assignment composition} \\
= & okay \Rightarrow okay' \wedge v' = f(e) & \text{design} \\
= & (\, \mathbf{true} \vdash v' = f(e) \,) & \text{assignment} \\
= & v := f(e)
\end{array}
$$

When program operators are applied to designs, the result is also a design. This follows from the laws below, for choice, conditional, sequence, and recursion. These are stated in [117] as theorems. We label them **T1**, **T2**, and **T3**, and add a simplified version of **T3**, which is mentioned in [117] and we call **T3′**.

A choice between two designs is guaranteed to terminate when they both do; since either of them may be chosen, either postcondition may be established.

T1 $(\, (P_1 \vdash Q_1) \sqcap (P_2 \vdash Q_2) \,) = (\, P_1 \wedge P_2 \vdash Q_1 \vee Q_2 \,)$

Exercise 8. Prove Law **T1**.

If the choice between two designs depends on a condition b, then so do the precondition and postcondition of the resulting design.

T2 $(\, (P_1 \vdash Q_1) \lhd b \rhd (P_2 \vdash Q_2) \,) = ((P_1 \lhd b \rhd P_2) \vdash (\, Q_1 \lhd b \rhd Q_2) \,)$

A sequence of designs $(P_1 \vdash Q_1)$ and $(P_2 \vdash Q_2)$ terminates when P_1 holds (that is, $\neg\, (\neg\, P_1 \,;\, \mathbf{true})$), and Q_1 is guaranteed to establish P_2 $(\neg\, (Q_1 \,;\, \neg\, P_2))$. On termination, the sequence establishes the composition of the postconditions.

T3 $(\, (P_1 \vdash Q_1) \,;\, (P_2 \vdash Q_2) \,)$
$= (\, (\neg\, (\neg\, P_1 \,;\, \mathbf{true}) \wedge \neg\, (Q_1 \,;\, \neg\, P_2)) \vdash (Q_1 \,;\, Q_2) \,)$

We have said nothing in our discussion of designs that requires a precondition to be a simple condition rather than a relation, and this fact complicates the statement of Law **T3**; if P_1 actually is a condition, then $\neg\, (\neg\, P_1 \,;\, \mathbf{true})$ can be simplified.

T3′ $(\, (p_1 \vdash Q_1) \,;\, (P_2 \vdash Q_2) \,) = (\, (p_1 \wedge \neg\, (Q_1 \,;\, \neg\, P_2)) \vdash (Q_1 \,;\, Q_2) \,)$

To understand the simplification, consider the following lemma.

Lemma 3 (condition-right-unit). *Abort is a right-unit for conditions.*

$p \,;\, \mathbf{true} = p$

A dual result is that abort is a left-unit for conditions on after-states: that is, $(\mathbf{true} \,;\, p') = p'$. We give two last results of this kind before we move on.

Lemma 4 (abort-condition).

$$\mathbf{true} \,;\, p \;=\; \exists\, v \bullet p$$

A special case of this lemma involves *okay* or, more generally, any boolean variable used as a condition.

Lemma 5 (abort-boolean). *Provided b is a boolean variable,*

$$\mathbf{true} \,;\, b \;=\; \mathbf{true}$$

A recursively defined design has as its body a function on designs; as such, it can be seen as a function on pre-post pairs (X, Y). Moreover, since the result of the function is itself a design, it can be written in terms of a pair of functions F and G, one for the precondition and one for the postcondition.

As the recursive design is executed, the precondition F is required to hold over and over again. The strongest recursive precondition so obtained has to be satisfied, if we are to guarantee that the recursion terminates. Similarly, the postcondition is established over and over again, in the context of the precondition. The weakest result that can possibly be obtained is that which can be guaranteed by the recursion.

T4 $\quad (\,\mu\, X, Y \bullet (\, F(X, Y) \vdash G(X, Y)\,)\,) = (\, P(Q) \vdash Q\,)$
where $P(Y) = (\,\nu X \bullet F(X, Y)\,)$ and $Q = (\,\mu\, Y \bullet P(Y) \Rightarrow G(P(Y), Y)\,)$

Further intuition comes from the realisation that we want the least refined fixed point of the pair of functions. That comes from taking the strongest precondition, since the precondition of every refinement must be weaker, and the weakest postcondition, since the postcondition of every refinement must be stronger.

Like the set of general alphabetised predicates, designs form a complete lattice. We have already presented the top and the bottom (miracle and abort).

$$\top_D \;\widehat{=}\; (\,\mathbf{true} \vdash \mathbf{false}\,) = \neg\, \mathit{okay}$$

$$\bot_D \;\widehat{=}\; (\,\mathbf{false} \vdash \mathbf{true}\,) = \mathbf{true}$$

Example 9 (Abort). All useful work is discarded once a program aborts. This is shown in the following derivation.

$$\begin{array}{lr}
\quad x := e \,;\, \bot_D & \text{assignment, bottom} \\
= (\,\mathbf{true} \vdash x' = e\,) \,;\, (\,\mathbf{false} \vdash \mathbf{true}\,) & \text{design sequence} \\
= \mathbf{true} \wedge \neg\, (x' = e \,;\, \neg\, \mathbf{false}) \vdash x' = e \,;\, \mathbf{true} & \text{relational calculus} \\
= \neg\, (x' = e \,;\, \mathbf{true}) \vdash \mathbf{true} & \text{relational calculus} \\
= \neg\, \mathbf{true} \vdash \mathbf{true} & \text{propositional calculus} \\
= \mathbf{false} \vdash \mathbf{true} & \text{bottom} \\
= \bot_D &
\end{array}$$

Abort, however, is not a right-zero for sequence, since it will not take over if it is preceded by a miraculous program.

The greatest lower-bound and the least upper-bound are established in the following theorem.

Theorem 1 (meets-and-joins).

$$\sqcap_i \bullet (P_i \vdash Q_i) = (\bigwedge_i \bullet P_i) \vdash (\bigvee_i \bullet Q_i)$$

$$\sqcup_i \bullet (P_i \vdash Q_i) = (\bigvee_i \bullet P_i) \vdash (\bigwedge_i \bullet P_i \Rightarrow Q_i)$$

As with the binary choice, the choice $\sqcap_i \bullet (P_i \vdash Q_i)$ over the set of designs $(P_i \vdash Q_i)$ terminates when all the designs do, and it establishes one of the possible postconditions. The least upper-bound models a form of choice that is conditioned by termination: only the terminating designs can be chosen. The choice terminates if any of the designs do, and the postcondition established is that of any of the terminating designs.

Designs are special kinds of relations, which in turn are special kinds of predicates, and so they can be combined with the propositional operators. A design can be negated, although the result is not itself a design.

Lemma 6 (not-design).

$$\neg (P \vdash Q) = (\mathit{okay} \wedge P \wedge (\mathit{okay}' \Rightarrow \neg Q))$$

If the postcondition of a design promises the opposite of its precondition, then the design is miraculous.

Law 69 (design-contradiction). $(P \vdash \neg P) = (P \vdash \mathbf{false})$

Proof.

$$\begin{array}{lr} P \vdash \neg P & \text{export-precondition} \\ = P \vdash P \wedge \neg P & \text{propositional calculus} \\ = P \vdash \mathbf{false} & \end{array}$$

Another way of characterising the set of designs is by imposing healthiness conditions on alphabetised predicates. Hoare & He identify four healthiness conditions that they consider of interest: ***H1*** to ***H4***. We discuss two of them.

4.1 *H1*: Unpredictability

A relation P is ***H1***-healthy if and only if $P = (\mathit{okay} \Rightarrow P)$. This means that observations cannot be made before the program has started. The idempotent corresponding to this healthiness condition is defined as

$$\mathbf{H1}(P) = \mathit{okay} \Rightarrow P$$

It is indeed an idempotent, since implication is idempotent in its first argument.

Law 70 (H1-idempotent). $\boldsymbol{H1} \circ \boldsymbol{H1} = \boldsymbol{H1}$

Proof.

$$
\begin{array}{lr}
\quad \boldsymbol{H1} \circ \boldsymbol{H1}(P) & \boldsymbol{H1} \\
= \mathit{okay} \Rightarrow (\mathit{okay} \Rightarrow P) & \text{propositional calculus} \\
= \mathit{okay} \wedge \mathit{okay} \Rightarrow P & \text{propositional calculus} \\
= \mathit{okay} \Rightarrow P & \boldsymbol{H1} \\
= \boldsymbol{H1}(P)
\end{array}
$$

*Example 10 (**H1** relations).* The following are examples of **H1** relations.

1. The relation **true**, since $(\mathit{okay} \Rightarrow \mathbf{true}) = \mathbf{true}$; it is also the design $\bot_D$: abort.
2. The relation $\neg\ \mathit{okay}$, since $(\mathit{okay} \Rightarrow \neg\ \mathit{okay}) = \neg\ \mathit{okay}$; it is also the design $\top_D$: miracle.
3. The relation $(\mathit{okay} \wedge x \neq 0 \Rightarrow x' < x)$, which, when started in a state where *okay* is true and $x \neq 0$, ensures that the after value of x is strictly less than its before value.
4. The design $(x \neq 0 \vdash x' < x)$, which, when started in a state where *okay* is true and $x \neq 0$, ensures *termination* and that the after value of x is strictly less than its before value.

Healthiness conditions give a way of imposing structure on a subset of relations, and **H1**-relations have some interesting algebraic properties. First, all **H1**-relations have a left zero.

Lemma 7 (H1-left-zero). *Provided P is **H1**-healthy,*

$$\mathbf{true}\ ;\ P = \mathbf{true}$$

All **H1**-relations have a left unit.

Lemma 8 (H1-left-unit). *Provided P is **H1**-healthy,*

$$\mathit{II}_D\ ;\ P\ =\ P$$

Finally, relations that have both left units and left zeros are also **H1**.

Lemma 9 (left-unit-zero-H1). *Provided P has a left unit and a left zero,*

$$P\ =\ (\mathit{okay} \Rightarrow P)$$

These three lemmas allow us to characterise **H1**-relations algebraically: they are exactly those relations that satisfy the left zero and left unit laws.

Theorem 2 (H1-healthiness).

$$(P = \boldsymbol{H1}(P))\ =\ ((\mathbf{true}\ ;\ P = \mathbf{true})\ \wedge\ (\mathit{II}_D\ ;\ P = P))$$

We conclude this section by investigating a few more of **H1**'s properties. It relates the two identities that we have seen so far.

Law 71 (II_D-*H1*-II). $\mathit{II}_D = \boldsymbol{H1}(\mathit{II})$

Proof.

$$
\begin{array}{lr}
\mathit{II}_D & \mathit{II}_D \\
= (\mathbf{true} \vdash \mathit{II}) & \text{design} \\
= (\mathit{okay} \Rightarrow \mathit{okay}' \wedge \mathit{II}) & \mathit{II} \\
= (\mathit{okay} \Rightarrow \mathit{okay}' \wedge \mathit{II} \wedge \mathit{okay}' = \mathit{okay}) & \text{propositional calculus} \\
= (\mathit{okay} \Rightarrow \mathit{II} \wedge \mathit{okay}' = \mathit{okay}) & \mathit{II} \\
= (\mathit{okay} \Rightarrow \mathit{II}) & \boldsymbol{H1} \\
= \mathit{II}
\end{array}
$$

H1 tells us that, try as we might, we simply cannot make an observation of the behaviour of a design until after it has started. A design with a rogue postcondition, such as $(\mathbf{true} \vdash (\neg\ \mathit{okay} \Rightarrow x' = 0))$, tries to violate ***H1***, but it cannot. We could simplify it by expanding the definition of a design, and then simplifying the result with propositional calculus. It is possible to avoid this expansion by applying ***H1*** directly to the postcondition.

Law 72 (design-post-*H1*). $(P \vdash Q) \ = \ (P \vdash \boldsymbol{H1}(Q))$

Proof.

$$
\begin{array}{lr}
P \vdash \boldsymbol{H1}(Q) & \boldsymbol{H1} \\
= P \vdash (\mathit{okay} \Rightarrow Q) & \text{design} \\
= \mathit{okay} \wedge P \Rightarrow \mathit{okay}' \wedge (\mathit{okay} \Rightarrow Q) & \text{propositional calculus} \\
= \mathit{okay} \wedge P \Rightarrow \mathit{okay}' \wedge Q & \text{design} \\
= P \vdash Q
\end{array}
$$

We can also push the application of ***H1*** in a postcondition through a negation.

Law 73 (design-post-not-*H1*). $(P \vdash \neg\ Q) \ = \ (P \vdash \neg\ \boldsymbol{H1}(Q))$

Proof.

$$
\begin{array}{lr}
P \vdash \neg\ \boldsymbol{H1}(Q) & \boldsymbol{H1} \\
= P \vdash \neg\ (\mathit{okay} \Rightarrow Q) & \text{propositional calculus} \\
= P \vdash \mathit{okay} \wedge \neg\ Q & \text{design} \\
= \mathit{okay} \wedge P \Rightarrow \mathit{okay}' \wedge \mathit{okay} \wedge \neg\ Q & \text{propositional calculus} \\
= \mathit{okay} \wedge P \Rightarrow \mathit{okay}' \wedge \neg\ Q & \text{design} \\
= P \vdash \neg\ Q
\end{array}
$$

H1 enjoys many other properties, some of which we see later in this chapter.

4.2 *H2*: Termination Always Possible

The second healthiness condition is $[\, P[\mathit{false}/\mathit{okay}'] \Rightarrow P[\mathit{true}/\mathit{okay}']\,]$. This means that if P is satisfied when okay' is *false*, it is also satisfied then okay' is *true*. In other words, P cannot *require* nontermination, so that termination is always a possibility.

*Example 11 (**H2** predicates).*

1. The design relations abort and miracle are both ***H2***, since they leave the value of okay' completely unconstrained.
2. The relation $(\mathit{okay}' \wedge (x' = 0))$ is ***H2***, since it insists on termination.
3. The design $(x \neq 0 \vdash x' < x)$ is ***H2***, since (a) if it is not started properly $(\neg\ \mathit{okay})$, or if $x = 0$, then it leaves okay' unconstrained; and (b) if it is started properly (okay) and $x \neq 0$, then it insists on termination.

If P is a predicate with okay' in its alphabet, we abbreviate $P[b/\mathit{okay}']$ as P^b, for boolean value b. Thus, P is ***H2***-healthy if and only if $[\, P^f \Rightarrow P^t\,]$, where f and t are used as abbreviations for **false** and **true**.

This healthiness condition may also be described in terms of an idempotent. For that, we define the following predicate.

Definition 1 (The idempotent J).

$$J \mathrel{\widehat{=}} (\mathit{okay} \Rightarrow \mathit{okay}') \wedge \mathit{II}_{rel}^{-okay}$$

J permits a change in the value of *okay*, while the remaining variables stay constant: if *okay* is changed, then it can be weakened, but not strengthened. We use $\mathit{II}_{rel}^{-okay}$ to denote the conjunction of equalities $v' = v$ for all variables v in the alphabet, except *okay*.

The relationship between J and ***H2*** (see Theorem 3) is based on an important property called J-split. As its name suggests, it divides a relation into two parts, but we must notice the asymmetry.

Lemma 10 (J-split). *Provided okay and okay' are in the alphabet of P,*

$$P\ ;\ J \ =\ P^f \vee (P^t \wedge \mathit{okay}')$$

The healthiness condition ***H2*** may now be expressed using J, as we show in the following theorem, which uses J-split.

Theorem 3 (*H2* equivalence). *There are two equivalent ways of characterising **H2**-healthy relations.*

$$(P = P\ ;\ J) \ =\ [\, P^f \Rightarrow P^t\,]$$

Based on this result, we use ***H2*** to refer to the function $\mathbf{H2}(P) = P;\ J$.

Interestingly, J is actually an ***H2***-healthy relation.

Lemma 11 (J is *H2*). *J is **H2**-healthy.*

$$J = \mathbf{H2}(J)$$

This lemma makes it easy to show that J really is an idempotent.

Law 74 (*H2*-idempotent). $\boldsymbol{H2} \circ \boldsymbol{H2} = \boldsymbol{H2}$

Proof.

$$
\begin{array}{lr}
\quad \boldsymbol{H2} \circ \boldsymbol{H2}(P) & \boldsymbol{H2} \\
= (P \,;\, J) \,;\, J & \text{associativity} \\
= P \,;\, (J \,;\, J) & \boldsymbol{H2} \\
= P \,;\, \boldsymbol{H2}(J) & J \ \boldsymbol{H2}\text{-healthy} \\
= P \,;\, J & \boldsymbol{H2} \\
= \boldsymbol{H2}(P) &
\end{array}
$$

As we see in the next two examples, the original formulation of ***H2*** is often easier to use in demonstrating that a relation is ***H2***; however, because the description based on J is an idempotent function, it has some interesting algebraic properties. First, we prove that a relation is ***H2*** using substitution.

*Example 12 (**H2**-substitution).*

$okay' \wedge (x' = 0)$ is ***H2***

Proof.

$$
\begin{array}{lr}
\quad (okay' \wedge (x' = 0))^f \Rightarrow (okay' \wedge (x' = 0))^t & \text{substitution} \\
= \mathbf{false} \wedge (x' = 0) \Rightarrow \mathbf{true} \wedge (x' = 0) & \text{propositional calculus} \\
= \mathbf{false} \Rightarrow (x' = 0) & \text{propositional calculus} \\
= \mathbf{true} &
\end{array}
$$

Now we prove that the same relation is ***H2*** using J.

*Example 13 (**H2**-J).*

$okay' \wedge (x' = 0)$ is ***H2***

Proof.

$$
\begin{array}{lr}
\quad okay' \wedge (x' = 0) \,;\, J & J\text{-splitting} \\
= (okay' \wedge (x' = 0))^f \vee ((okay' \wedge (x' = 0))^t \wedge okay') & \text{substitution} \\
= (\mathbf{false} \wedge (x' = 0)) \vee (\mathbf{true} \wedge (x' = 0) \wedge okay') & \text{propositional calculus} \\
= \mathbf{false} \vee ((x' = 0) \wedge okay') & \text{propositional calculus} \\
= okay' \wedge (x' = 0) &
\end{array}
$$

We said at the beginning of this section that we would characterise the space of designs using our healthiness conditions, and we can now do this. If a relation is both ***H1*** and ***H2***-healthy, then it is also a design.

Lemma 12 (*H1*-*H2* is a design). *If P is a relation that is both **H1** and **H2**-healthy, then it can be expressed as the design $\neg\, P^f \vdash P^t$.*

This result also holds in the other direction, as we have already illustrated in the examples. First, we establish that designs are ***H2*** relations.

Lemma 13 (Designs are *H2*). *Provided P and Q do not have okay and okay′ in their alphabets,*

$$[\,(\,P \vdash Q\,)^f \Rightarrow (\,P \vdash Q\,)^t\,]$$

It is obvious that all designs are ***H1***: the proof is a nice little exercise. So, we have the following theorem.

Theorem 4. *A relation with alphabet including okay and okay′ is a design exactly when it is both **H1** and **H2**-healthy.*

An important property of healthiness conditions is commutativity. For example, ***H1*** and ***H2*** commute.

Law 75 (commutativity-*H2*-*H1*). $\boldsymbol{H2} \circ \boldsymbol{H1} = \boldsymbol{H1} \circ \boldsymbol{H2}$

Proof.

$$\begin{array}{lr}
\quad \boldsymbol{H1} \circ \boldsymbol{H2}(P) & \boldsymbol{H1}, \boldsymbol{H2} \\
= \mathit{okay} \Rightarrow P \,;\, J & \text{propositional calculus} \\
= \neg\, \mathit{okay} \;\vee\; P \,;\, J & \text{miracle is } \boldsymbol{H2} \\
= \boldsymbol{H2}(\neg\, \mathit{okay}) \;\vee\; P \,;\, J & \boldsymbol{H2} \\
= \neg\, \mathit{okay} \,;\, J \;\vee\; P \,;\, J & \text{relational calculus} \\
= (\neg\, \mathit{okay} \vee P) \,;\, J & \text{propositional calculus} \\
= (\mathit{okay} \Rightarrow P) \,;\, J & \boldsymbol{H1}, \boldsymbol{H2} \\
= \boldsymbol{H2} \circ \boldsymbol{H1}(P)
\end{array}$$

This means that we can apply ***H1*** and ***H2*** independently to make a relation healthy. The result is a relation that is both ***H1*** and ***H2***-healthy, and, moreover, it is the same no matter in which order we applied ***H1*** and ***H2***.

5 Reactive Processes

A reactive program interacts with its environment, which can include other programs as well as the users of the system. A reactive program's behaviour cannot be characterised by its final state alone; we need to record information about interactions with the environment. Actually, many reactive programs never terminate, and so do not even have a final state; their whole purpose is to interact with the environment. Each interaction, whether it be a synchronisation or a communication, is an event.

To model a reactive process, we use the *okay* variable and three extra observational variables: *tr*, *ref*, and *wait*, and their dashed counterparts. The finite sequences *tr* and *tr′* record the events that occurred up to the moment of the observation. The sets *ref* and *ref′* record events that may be refused. The variables

wait and $wait'$ are boolean; $wait'$ records whether the process has terminated or is in an intermediate state awaiting further interaction with the environment.

When $okay'$ is true for a design, it means that the design has reached a final state. The same is true for a reactive process that has $okay'$ true and $wait'$ false. If both $okay'$ and $wait'$ are true, then it means that the reactive process has reached an intermediate state. If $okay'$ is false, then it means that the process has reached neither an intermediate nor a final state. So, the meaning of $okay'$ is the same in both theories: when it is true, it indicates that a stable state has been reached; when it is false it indicates the opposite. The difference is that the notion of a stable state is richer for reactive processes, as it includes intermediate states. In view of this, we change our terminology: instead of saying that a design has *aborted*, we say that a process has *diverged*.

Of course, these comments apply to *okay* as well. When it is true, it means that the process is in a stable state. This, however, may be an intermediate of another process that is currently executing. The process only really starts when *wait* is false.

In summary, there are three distinct observations that may be made of *okay* and *wait*.

$okay \wedge \neg\, wait$	started in a stable state
$okay \wedge wait$	not started, but in a stable state
$\neg\, okay$	not started, but in an unstable state

Similarly, there are three observations that may be made of the final values of these two variables.

$okay' \wedge \neg\, wait'$	terminated
$okay' \wedge wait'$	in an intermediate state
$\neg\, okay'$	in an unstable state

With these observations, it is clear that a reactive process is properly started if it is initiated in a state with *wait* false; that is, if its predecessor has terminated.

We often want to refer to a predicate $P[false/wait]$, which we abbreviate as P_f. Combining this with our earlier notation, P_f^t describes a reactive process P that was properly started, and has not diverged. This substitution does not disturb healthiness conditions that do not mention *wait* and $okay'$, such as ***H1***.

Law 76 (*H1*-*wait*-$okay'$). $(\boldsymbol{H1}(P))_b^c \;=\; \boldsymbol{H1}(P_b^c)$

Not every relation is a reactive process; just like designs, some healthiness conditions need to be imposed. Before we investigate them, however, we give a simple example of a reactive process.

5.1 Reactive *II*

A reactive process is a relation with all eight observational variables in its alphabet. Perhaps the simplest example is the reactive Π, which is defined as follows.

Definition 2 (II_{rea}).

$$\begin{array}{rcl} \mathit{II}_{rea} & \widehat{=} & \neg\ okay \wedge tr \leq tr' \\ & & \vee \\ & & okay' \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref \end{array}$$

The behaviour of II_{rea} depends on its initial state: if it was an unstable state ($\neg\ okay$), then the first disjunct applies; otherwise, the second disjunct applies. In the first case, the predicate $tr \leq tr'$ requires that tr is a prefix of tr'. The trace tr contains a record of all the events that occurred before the initial observation of II_{rea}; in the final observation, the trace tr' must be an extension of tr: the process cannot change history by modifying the sequence of events that have already occurred. In the second case, the initial state was stable, and the behaviour is the same, regardless of whether the process was started or not: all variables must remain constant.

Alternative definitions of II_{rea} can be formulated. For example, it can be defined in terms of the relational II and in terms of the conditional.

Law 77 (II_{rea}-II_{rel}). $\mathit{II}_{rea} = \neg\ okay \wedge tr \leq tr' \ \vee\ \mathit{II}_{rel}$

Law 78 (II_{rea}-II_{rel}-conditional). $\mathit{II}_{rea} = \mathit{II}_{rel} \lhd okay \rhd tr \leq tr'$

The law below states that in a stable state, II_{rea} is just like II_{rel}.

Law 79 ($okay$-II_{rea}-II_{rel}). $okay \wedge \mathit{II}_{rea} = okay \wedge \mathit{II}_{rel}$

As an obvious consequence, II_{rea} is a unit for sequence in a stable state. Of course, in general it is not an identity, since in an unstable state it guarantees only that the trace is either left untouched or extended.

Law 80 ($okay$-II_{rea}-sequence-unit). $okay \wedge \mathit{II}_{rea}\ ;\ P = okay \wedge P$

Exercise 9. Prove Law 80.

Is II_{rea} a design? Well, it is certainly ***H2***-healthy.

Law 81 (II_{rea}-*H2*). $\mathit{II}_{rea} = \mathbf{H2}(\mathit{II}_{rea})$

Proof.

$$\begin{array}{rl} & \mathbf{H2}(\mathit{II}_{rea}) \\ = & \mathit{II}_{rea}{}^{f} \vee (\mathit{II}_{rea}{}^{t} \wedge okay') \\ = & (\neg\ okay \wedge tr \leq tr') \vee (\neg\ okay \wedge tr \leq tr' \ \vee\ \mathit{II}_{rel}) \wedge okay' \\ = & (\neg\ okay \wedge tr \leq tr') \ \vee\ (\neg\ okay \wedge tr \leq tr' \wedge okay') \vee \mathit{II}_{rel}{}^{t} \wedge okay' \\ = & \neg\ okay \wedge tr \leq tr' \ \vee\ \mathit{II}_{rel}{}^{t} \wedge okay' \\ = & \neg\ okay \wedge tr \leq tr' \ \vee\ okay' \wedge \mathit{II}_{rel} \\ = & \mathit{II}_{rea} \end{array} \qquad \begin{array}{r} J\text{-splitting} \\ \mathit{II}_{rea} \\ \text{propositional calculus} \\ \text{absorption} \\ \text{Leibniz} \\ \mathit{II}_{rea} \end{array}$$

Although, being $\boldsymbol{H2}$, $\mathbb{I}_{rea}$ is half-way to being a design, it is not $\boldsymbol{H1}$-healthy. This is because its behaviour when *okay* is false is not arbitrary as $\boldsymbol{H1}$ requires: the restriction on the traces still applies. In fact, the healthiness condition $\boldsymbol{H1}$ relates the two identities in the following way.

Law 82 ($\boldsymbol{H1}$-$\mathbb{I}_{rea}$-$\mathbb{I}_{rel}$). $\boldsymbol{H1}(\mathbb{I}_{rea}) = \boldsymbol{H1}(\mathbb{I}_{rel})$

So $\mathbb{I}_{rea}$ fails to be a design; in fact, *no* reactive process is a design, although as we shall see, they can all be expressed in terms of a design. So, the theory of reactive processes is a subtheory of the theory of relations that is distinct from the theory of designs. The question is: which relations are reactive processes? This answered by three healthiness conditions.

5.2 *R1*

Time travel is practically a weekly event in *Star Trek* and *Doctor Who*, and it is certainly an interesting activity that has much to offer the curious mind, but it will be outlawed by our first reactive healthiness condition, $\boldsymbol{R1}$. This requires that a relation cannot change the trace of events that have already occurred. The idempotent is as follows.

$$\boldsymbol{R1}(P) = P \wedge tr \leq tr'$$

We already saw this in the definition of $\mathbb{I}_{rea}$: if its initial observation is made in an unstable state (*okay* is false), then the trace in its final observation will be an extension of the initial trace; if the initial observation is made in a stable state, then the trace is kept constant. We have that $\boldsymbol{R1}$ is an idempotent because of idempotency of conjunction.

Law 83 (R1-idempotent). $\boldsymbol{R1} \circ \boldsymbol{R1} = \boldsymbol{R1}$

The simplicity of $\boldsymbol{R1}$ leads to many obvious algebraic laws. For example, it distributes through both conjunction and disjunction, and because it is defined by conjunction, its scope may be extended over other conjunctions. As a consequence of these laws, $\boldsymbol{R1}$ distributes through the conditional and the unhealthy effects of negation can be swiftly cured. Finally, substitution for *wait* and for *okay*′ both distribute through $\boldsymbol{R1}$.

Law 84 ($\boldsymbol{R1}$-$\wedge$). $\boldsymbol{R1}(P \wedge Q) = \boldsymbol{R1}(P) \wedge \boldsymbol{R1}(Q)$

Law 85 ($\boldsymbol{R1}$-$\vee$). $\boldsymbol{R1}(P \vee Q) = \boldsymbol{R1}(P) \vee \boldsymbol{R1}(Q)$

Law 86 ($\boldsymbol{R1}$-extend-$\wedge$). $\boldsymbol{R1}(P) \wedge Q = \boldsymbol{R1}(P \wedge Q)$

Law 87 ($\boldsymbol{R1}$-conditional). $\boldsymbol{R1}(P \lhd b \rhd Q) = \boldsymbol{R1}(P) \lhd b \rhd \boldsymbol{R1}(Q)$

Law 88 ($\boldsymbol{R1}$-negate-$\boldsymbol{R1}$). $\boldsymbol{R1}(\neg\, \boldsymbol{R1}(P)) = \boldsymbol{R1}(\neg\, P)$

Law 89 ($\boldsymbol{R1}$-*wait*-*okay*′). $(\boldsymbol{R1}(P))^c_b = \boldsymbol{R1}(P^c_b)$

Both the relational and the reactive identities are ***R1***-healthy.

Law 90 (II_{rel}-*R1*). $\mathit{II}_{rel} = \boldsymbol{R1}(\mathit{II}_{rel})$

Exercise 10. Prove Law 90.

The fact that II_{rel} is ***R1*** is helpful in proving that II_{rea} is too.

Law 91 (II_{rea}-*R1*). $\mathit{II}_{rea} = \boldsymbol{R1}(\mathit{II}_{rea})$

Proof.

$$
\begin{array}{lr}
\quad \boldsymbol{R1}(\mathit{II}_{rea}) & \mathit{II}_{rea}\text{-conditional} \\
= \boldsymbol{R1}(\mathit{II}_{rel} \lhd \mathit{okay} \rhd \mathit{tr} \leq \mathit{tr}') & \boldsymbol{R1}\text{-conditional} \\
= \boldsymbol{R1}(\mathit{II}_{rel}) \lhd \mathit{okay} \rhd \boldsymbol{R1}(\mathbf{true}) & \text{propositional calculus, } \boldsymbol{R1} \\
= \boldsymbol{R1}(\mathit{II}_{rel}) \lhd \mathit{okay} \rhd \mathit{tr} \leq \mathit{tr}' & \mathit{II}_{rel}\text{-}\boldsymbol{R1} \\
= \mathit{II}_{rel} \lhd \mathit{okay} \rhd \mathit{tr} \leq \mathit{tr}' & \mathit{II}_{rea} \text{ conditional} \\
= \mathit{II}_{rea}
\end{array}
$$

By applying any of the program operators to an ***R1***-healthy process, we get another ***R1***-healthy process. That is, ***R1*** is closed under conjunction, disjunction, conditional, and sequence.

Theorem 5. *Provided P and Q are* ***R1****-healthy,*

$$
\begin{array}{ll}
\boldsymbol{R1}(P \wedge Q) = P \wedge Q & \boldsymbol{R1}\text{-}\wedge\text{-}\mathit{closure} \\
\boldsymbol{R1}(P \vee Q) = P \vee Q & \boldsymbol{R1}\text{-}\vee\text{-}\mathit{closure} \\
\boldsymbol{R1}(P \lhd b \rhd Q) = P \lhd b \rhd Q & \boldsymbol{R1}\text{-}\mathit{conditional\text{-}closure} \\
\boldsymbol{R1}(P \,;\, Q) = P \,;\, Q & \boldsymbol{R1}\text{-}\mathit{sequence\text{-}closure}
\end{array}
$$

The distribution properties are stronger than the closure properties, and this is clear from the fact that the proofs of ***R1***-$\wedge$-*closure*, ***R1***-$\vee$-*closure*, and ***R1***-*conditional-closure* follow immediately from Laws 84, 85, and 87, respectively. Law ***R1***-*sequence-closure* is rather more interesting, since ***R1*** does not distribute through sequence. First we note a result from the relational calculus that relates sequence and transitive relations.

Lemma 14 (sequence-transitive-relation). *Provided $\preceq$ is a transitive relation,*

$$[\,((P \wedge x \preceq x') \,;\, (Q \wedge x \preceq x')) \Rightarrow x \preceq x'\,]$$

This establishes that if the first program in the sequence assigns to x a value that is related to its original value by the transitive relation $\preceq$, and the second program takes this (intermediate) value and assigns to x a value that is related to it, then transitivity allows us to conclude that the original and final values of x are related by $\preceq$.

Exercise 11. Prove Lemma 14.

In our case, the transitive relation in which we are interested in sequence prefixing.

Proof of ***R1****-sequence-closure.*

$$
\begin{array}{ll}
& P \,;\, Q & \text{assumption: } P \text{ and } Q \text{ both } \boldsymbol{R1} \\
= & \boldsymbol{R1}(P) \,;\, \boldsymbol{R1}(Q) & \boldsymbol{R1}\text{, twice} \\
= & P \wedge tr \leq tr' \,;\, Q \wedge tr \leq tr' & \text{sequence-transitive-relation} \\
= & (P \wedge tr \leq tr' \,;\, Q \wedge tr \leq tr') \wedge tr \leq tr' & \boldsymbol{R1}\text{, three times} \\
= & \boldsymbol{R1}(\boldsymbol{R1}(P) \,;\, \boldsymbol{R1}(Q)) & \text{assumption: } P \text{ and } Q \text{ both } \boldsymbol{R1} \\
= & \boldsymbol{R1}(P \,;\, Q)
\end{array}
$$

$\mathbb{I}_{rea}$ is an ***R1*** relation, but as we have seen it is not ***H1***. Of course, we can make it ***H1*** by applying the healthiness condition, but then it is no longer ***R1***. If we apply ***R1*** once again, we get back to where we started.

Law 92 ($\mathbb{I}_{rea}$**-*R1*-*H1*).** $\mathbb{I}_{rea} = \boldsymbol{R1} \circ \boldsymbol{H1}(\mathbb{I}_{rea})$

Proof.

$$
\begin{array}{ll}
& \boldsymbol{R1} \circ \boldsymbol{H1}(\mathbb{I}_{rea}) & \boldsymbol{H1}\text{-}\mathbb{I}_{rea}\text{-}\mathbb{I}_{rel} \\
= & \boldsymbol{R1} \circ \boldsymbol{H1}(\mathbb{I}_{rel}) & \boldsymbol{H1} \\
= & \boldsymbol{R1}(\mathit{okay} \Rightarrow \mathbb{I}_{rel}) & \text{propositional calculus} \\
= & \boldsymbol{R1}(\neg\, \mathit{okay} \vee \mathbb{I}_{rel}) & \boldsymbol{R1}\text{-}\vee\text{-distribution} \\
= & \boldsymbol{R1}(\neg\, \mathit{okay}) \vee \boldsymbol{R1}(\mathbb{I}_{rel}) & \mathbb{I}_{rel}\text{-}\boldsymbol{R1} \\
= & \boldsymbol{R1}(\neg\, \mathit{okay}) \vee \mathbb{I}_{rel} & \boldsymbol{R1} \\
= & \neg\, \mathit{okay} \wedge tr \leq tr' \vee \mathbb{I}_{rel} & \mathbb{I}_{rea} \\
= & \mathbb{I}_{rea}
\end{array}
$$

Law 92 shows that ***R1*** and ***H1*** are not independent: they do not commute; but ***R1*** does commute with ***H2***.

Law 93 (*R1*-*H2*-commutativity). $\boldsymbol{R1} \circ \boldsymbol{H2} = \boldsymbol{H2} \circ \boldsymbol{R1}$

Proof.

$$
\begin{array}{ll}
& \boldsymbol{H2} \circ \boldsymbol{R1}(P) & \boldsymbol{R1} \\
= & \boldsymbol{H2}(P \wedge tr \leq tr') & \boldsymbol{H2}\text{-}\wedge\text{-non-}\mathit{okay} \\
= & \boldsymbol{H2}(P) \wedge tr \leq tr' & \boldsymbol{R1} \\
= & \boldsymbol{R1} \circ \boldsymbol{H2}(P)
\end{array}
$$

The space described by applying ***R1*** to designs is a complete lattice because ***R1*** is monotonic. The relevance of this fact is made clear in the next section.

5.3 *R2*

The trace of a reactive process is an observation that is useful in describing the behaviour of concurrency and communication in reactive systems. We do

not imagine that any programmer would want to include such a variable in a real program; the overhead of keeping an accurate record of all events that have occurred since a program was started would be huge, and there is no need to keep it anyway. Rather, tr and tr' play similar roles to *okay* and *okay*$'$: they are devices that allow us to give an account of the behaviour of programming language constructs.

Designs are sensitive to the initial value of *okay*: the design cannot be started unless *okay* is true. But there is no obvious reason why a reactive process should be sensitive to the initial value of tr; in fact, none of the programming language constructs that we will introduce are sensitive to its value. The only purpose given to the trace is to provide an abstract view of program behaviour. For these reasons, we introduce a second healthiness condition that requires reactive processes to be insensitive to the value of tr.

There are two alternative formulations for this healthiness condition. Intuitively, they each establish that a process description should not rely on the history that passed before its activation, and should restrict only the new events to be recorded since the last observation. These are the events in $tr' - tr$.

The first formulation requires that P is not changed if tr is replaced by an arbitrary value. Of course, if tr is changed, then a corresponding change must be made to tr', otherwise all chance of ***R1*** healthiness will be compromised.

$$\boldsymbol{R2a}(P(tr, tr')) = \sqcap_s \bullet P(s, s \frown (tr' - tr))$$

The second formulation requires that P is not changed if the value of tr is taken to be the empty sequence.

$$\boldsymbol{R2b}(P(tr, tr')) = P(\langle\rangle, tr' - tr)$$

R2a and ***R2b*** are different functions. To see this, compare what happens when each function is applied to the relation $tr = \langle a \rangle$.

$$\begin{array}{ll} & \boldsymbol{R2a}(tr = \langle a \rangle) \\ = & \sqcap s \bullet s = \langle a \rangle \\ = & \mathbf{true} \sqcap \mathbf{false} \\ = & \mathbf{true} \end{array} \qquad \begin{array}{ll} & \boldsymbol{R2b}(tr = \langle a \rangle) \\ = & (tr = \langle a \rangle)[\langle\rangle, tr' - tr / tr, tr'] \\ = & (\langle\rangle = \langle a \rangle) \\ = & \mathbf{false} \end{array}$$

Even though they are different functions, they do have much in common. First, every ***R2b***-healthy relation is also ***R2a***-healthy; that is, for every relation P, $\boldsymbol{R2b}(P)$ is a fixed point of ***R2a***.

Law 94 (*R2b*-*R2a*). $\boldsymbol{R2b} = \boldsymbol{R2a} \circ \boldsymbol{R2b}$

Proof.

$$\begin{array}{ll} & \boldsymbol{R2a} \circ \boldsymbol{R2b}(P(tr, tr')) & \boldsymbol{R2b} \\ = & \boldsymbol{R2a}(P(\langle\,\rangle, tr' - tr)) & \boldsymbol{R2a} \end{array}$$

$$
\begin{array}{ll}
= \sqcap s \bullet P(\langle\rangle, tr' - tr)(s, s \frown (tr' - tr)) & \text{substitution} \\
= \sqcap s \bullet P(\langle\rangle, s \frown (tr' - tr) - s) & \text{property of } - \\
= \sqcap s \bullet P(\langle\rangle, tr' - tr) & \text{property of } \sqcap \\
= P(\langle\rangle, tr' - tr) & \boldsymbol{R2b} \\
= \boldsymbol{R2b}(P) &
\end{array}
$$

Similarly, every ***R2a***-healthy relation is also ***R2b***-healthy; that is, for every relation P, $\boldsymbol{R2a}(P)$ is a fixed point of ***R2b***.

Law 95 (*R2a*-*R2b*). $\boldsymbol{R2a} = \boldsymbol{R2b} \circ \boldsymbol{R2a}$

Proof.

$$
\begin{array}{ll}
\quad \boldsymbol{R2b} \circ \boldsymbol{R2a}(P(tr, tr')) & \boldsymbol{R2a} \\
= \boldsymbol{R2b}(\sqcap s \bullet P(s, s \frown (tr' - tr))) & \boldsymbol{R2b} \\
= (\sqcap s \bullet P(s, s \frown (tr' - tr)))(\langle\rangle, tr' - tr) & \text{substitution} \\
= \sqcap s \bullet P(s, s \frown (tr' - tr) - \langle\rangle) & \text{property of } - \\
= \sqcap s \bullet P(s, s \frown (tr' - tr)) & \boldsymbol{R2a} \\
= \boldsymbol{R2a}(P) &
\end{array}
$$

Laws 94 and 95 show us that ***R2a*** and ***R2b*** have the same image; that is, they characterise the same set of healthy predicates. We adopt ***R2b*** as our second healthiness condition for reactive processes, and actually refer to it as ***R2***.

$$\boldsymbol{R2} = \boldsymbol{R2b}$$

Not all properties of ***R2b*** that we prove in the sequel hold for ***R2a***; so this is an important point.

The healthiness condition ***R2*** is an idempotent.

Law 96 (R2-idempotent). $\boldsymbol{R2} \circ \boldsymbol{R2} = \boldsymbol{R2}$

Again, the programming operators are closed with respect to ***R2***. For the conditional, we have a result for quite specific conditions. For brevity, we omit proofs.

Theorem 6. *Provided P and Q are* ***R2****-healthy, and tr and tr' are not in the alphabet of b,*

$$
\begin{array}{ll}
\boldsymbol{R2}(P \wedge Q) = P \wedge Q & \boldsymbol{R1}\text{-}\wedge\text{-closure} \\
\boldsymbol{R2}(P \vee Q) = P \vee Q & \boldsymbol{R1}\text{-}\vee\text{-closure} \\
\boldsymbol{R2}(P \lhd tr' = tr \rhd Q) = P \lhd tr' = tr \rhd Q & \boldsymbol{R2}\text{-conditional-closure-1} \\
\boldsymbol{R2}(P \lhd b \rhd Q) = P \lhd b \rhd Q & \boldsymbol{R2}\text{-conditional-closure-2} \\
\boldsymbol{R2}(P \,;\, Q) = P \,;\, Q & \boldsymbol{R2}\text{-sequence-closure}
\end{array}
$$

Conditionals whose condition involves tr or tr' are problematic, but as shown above, the particular condition $tr = tr'$ does not hamper distribution.

Our proof of Law ***R2*-sequence-closure** is based on a suggestion due to Chen Yifeng. ***R2*** does not distribute through the sequence $P \,;\, Q$ because it cannot constrain the hidden value of the trace that exists between the behaviours of P and Q. For example, we consider the sequence below.

$$tr' = tr \frown \langle\, a \,\rangle;\ \text{last}\ \ tr = a \wedge tr \leq tr'$$

It is an ***R2*** process, and so it is not changed by an application of ***R2***. The second process, however, is not ***R2*** as it relies on a particular property of the initial value of tr; namely, that its last element is a. If we apply ***R2*** to it, we get **false** as a result. Therefore,

$$\boldsymbol{R2}(tr' = tr \frown \langle\, a \,\rangle);\ \boldsymbol{R2}(\text{last}\ \ tr = a \wedge tr \leq tr')$$

is also **false**.

Exercise 12. Give an algebraic proof that the sequence above is ***R2***-healthy, or, in other words, $\boldsymbol{R2}(tr' = tr \frown \langle\, a \,\rangle;\ \text{last}\ \ tr = a \wedge tr \leq tr')$ is equal to $tr' = tr \frown \langle\, a \,\rangle;\ \text{last}\ \ tr = a \wedge tr \leq tr'$ itself, and that

$$\boldsymbol{R2}(tr' = tr \frown \langle\, a \,\rangle);\ \boldsymbol{R2}(\text{last}\ \ tr = a \wedge tr \leq tr') = \mathbf{false}$$

The proof below for ***R2*-sequence-closure** is based on the Laws *left-one-point* and *right-one-point* for sequences.

Proof of ***R2*-sequence-closure**.

$$\begin{array}{lr}
\quad \boldsymbol{R2}(P(tr, tr') \,;\, Q(tr, tr')) & \text{sequence, predicate calculus} \\
= \boldsymbol{R2}(\, P(tr, tr'_0) \,;\, Q(tr_0, tr')\,) & \boldsymbol{R2} \\
= (\, P(tr, tr'_0) \,;\, Q(tr_0, tr')\,)(\langle\rangle, tr' - tr) & \text{substitution} \\
= P(\langle\rangle, tr'_0) \,;\, Q(tr_0, tr' - tr) & \text{assumption: } Q \text{ is } \boldsymbol{R2} \\
= P(\langle\rangle, tr'_0) \,;\, Q(\langle\rangle, tr' - tr)(tr_0, tr' - tr) & \text{substitution} \\
= P(\langle\rangle, tr'_0) \,;\, Q(\langle\rangle, tr' - tr - tr_0) & \text{sequence property} \\
= P(\langle\rangle, tr'_0) \,;\, Q(\langle\rangle, tr' - (tr \frown tr_0)) & \text{substitution} \\
= P(\langle\rangle, tr'_0) \,;\, Q(\langle\rangle, tr' - tr)[tr \frown tr_0/tr] & \text{left-one-point} \\
= P(\langle\rangle, tr'_0) \,;\, tr' = tr \frown tr_0 \,;\, Q(\langle\rangle, tr' - tr) & \text{sequence property} \\
= P(\langle\rangle, tr'_0) \,;\, tr_0 = tr' - tr \,;\, Q(\langle\rangle, tr' - tr) & \text{right-one-point} \\
= P(\langle\rangle, tr' - tr) \,;\, Q(\langle\rangle, tr' - tr) & \text{assumption: } P \text{ and } Q \text{ are } \boldsymbol{R2} \\
= P \,;\, Q &
\end{array}$$

A by-product of the above proof is the following law.

Law 97 (*R2* composition). $\boldsymbol{R2}(P \,;\, \boldsymbol{R2}(Q)) \;=\; \boldsymbol{R2}(P) \,;\, \boldsymbol{R2}(Q)$

Since ***R2*** constrains only tr and tr', substitution for $wait$ and $okay'$ distribute through its application.

Law 98 (*R2-wait-okay′*). $(\boldsymbol{R2}(P))^c_b = \boldsymbol{R2}(P^c_b)$

If J (see Definition 1) is lifted to an alphabet containing the reactive observations, then it keeps the trace constant. It is therefore ***R2***-healthy.

Law 99 (*J-R2*). $J = \boldsymbol{R2}(J)$

R2 is independent from ***H1***, ***H2***, and ***R1***: it commutes with each of them.

Law 100 (commutativity-*R2-H1*). $\boldsymbol{R2} \circ \boldsymbol{H1} = \boldsymbol{H1} \circ \boldsymbol{R2}$

Law 101 (commutativity-*R2-H2*). $\boldsymbol{R2} \circ \boldsymbol{H2} = \boldsymbol{H2} \circ \boldsymbol{R2}$

Proof.

$$
\begin{array}{lr}
\quad \boldsymbol{R2} \circ \boldsymbol{H2}(P) & \boldsymbol{H2} \\
= \boldsymbol{R2}(P \,;\, J) & J\ \boldsymbol{R2} \\
= \boldsymbol{R2}(P \,;\, \boldsymbol{R2}(J)) & \boldsymbol{R2} \text{ composition} \\
= \boldsymbol{R2}(P) \,;\, \boldsymbol{R2}(J) & J\ \boldsymbol{R2} \\
= \boldsymbol{R2}(P) \,;\, J & \boldsymbol{H2} \\
= \boldsymbol{H2} \circ \boldsymbol{R2}(P)
\end{array}
$$

Law 102 (commutativity-*R2-R1*). $\boldsymbol{R2} \circ \boldsymbol{R1} = \boldsymbol{R1} \circ \boldsymbol{R2}$

Proof.

$$
\begin{array}{lr}
\quad \boldsymbol{R2} \circ \boldsymbol{R1}(P(tr, tr')) & \boldsymbol{R1}, \boldsymbol{R2} \\
= (P \wedge tr \leq tr')(\langle\rangle, tr' - tr) & \text{substitution} \\
= P(\langle\rangle, tr' - tr) \wedge \langle\rangle \leq tr' - tr & \leq \text{ and } - \\
= P(\langle\rangle, tr' - tr) \wedge tr \leq tr' & \boldsymbol{R1}, \boldsymbol{R2} \\
= \boldsymbol{R1} \circ \boldsymbol{R2}(P(tr, tr'))
\end{array}
$$

The space of relations produced by applying ***R2*** to designs is again a complete lattice, since ***R2*** is also monotonic.

5.4 *R3*

The third healthiness condition makes relational composition behave like a program sequence. To see its relevance, consider the process

$$P \;=\; okay' \wedge wait' \wedge tr' = tr$$

P forever occupies a state that is both stable and waiting for interaction with the environment, but none ever comes, since the trace never changes. What are we to make of the sequence $P \,;\, Q$, where

$$Q \;=\; okay' \wedge \neg\, wait' \wedge tr' = tr \frown \langle a \rangle$$

Q immediately terminates, having added an a event to the trace? The relational composition ignores the behaviour of P.

$$
\begin{array}{ll}
P \,;\, Q & \text{sequence} \\
= \exists\, okay_0, wait_0, tr_0, ref_0 \bullet & P \text{ and } Q \\
\quad P[okay_0, wait_0, tr_0, ref_0 / okay', wait', tr', ref'] \wedge & \\
\quad Q[okay_0, wait_0, tr_0, ref_0 / okay, wait, tr, ref] & \\
= \exists\, okay_0, wait_0, tr_0, ref_0 \bullet & \text{predicate calculus} \\
\quad okay_0 \wedge wait_0 \wedge tr_0 = tr \wedge & \\
\quad okay' \wedge \neg\, wait' \wedge tr' = tr_0 \frown \langle a \rangle & \\
= \exists\, ref_0 \bullet okay' \wedge \neg\, wait' \wedge tr' = tr \frown \langle a \rangle & \text{predicate calculus} \\
= okay' \wedge \neg\, wait' \wedge tr' = tr \frown \langle a \rangle & Q \\
= Q &
\end{array}
$$

We expect quite the opposite: that $P \,;\, Q = P$. If P is forever waiting, then $P \,;\, Q$ should be forever waiting *in the same state*. We formalise this requirement as a healthiness condition, ***R3***.

$$\boldsymbol{R3}(P) = (\mathbb{I}_{rea} \lhd wait \rhd P)$$

An ***R3***-healthy process does not start until its predecessor has terminated. Now we have that $P \,;\, \boldsymbol{R3}(Q) = P$, as we wanted.

Exercise 13. Prove that $P \,;\, \boldsymbol{R3}(Q) = P$, where P and Q are the processes defined above.

The following laws characterise the behaviour of ***R3*** processes in particular circumstances. ***R3*** depends on the *wait* observation, so substitution for that variable cannot distribute through the healthiness condition. Instead, it serves to simplify ***R3***'s conditional. If *true* is substituted, then the result is $\mathbb{I}_{rea}$, but with the substitution applied to that as well. On the other hand, if *false* is substituted for *wait* in $\boldsymbol{R3}(P)$, then the result is simply P, again with the substitution applied. Substitution for $okay'$ interferes with $\mathbb{I}_{rea}$, and so does not distribute through its application.

Law 103 (*R3*-*wait*-true). $(\boldsymbol{R3}(P))_t = (\mathbb{I}_{rea})_t$

Law 104 (*R3*-not-*wait*-false). $(\boldsymbol{R3}(P))_f = P_f$

Law 105 (*R3*-$okay'$). $(\boldsymbol{R3}(P))^c = ((\mathbb{I}_{rea})^c \lhd wait \rhd P^c)$

Closure properties are also available for ***R3***.

Theorem 7. *Provided P and Q are* ***R3****,*

$$
\begin{array}{ll}
\boldsymbol{R3}(P \wedge Q) = P \wedge Q & \textit{R3-}\wedge\textit{-closure} \\
\boldsymbol{R3}(P \vee Q) = P \vee Q & \textit{R3-}\vee\textit{-closure} \\
\boldsymbol{R3}(P \lhd c \rhd Q) = P \lhd c \rhd Q & \textit{R3-conditional-closure}
\end{array}
$$

For sequence, we actually require that one of the processes is ***R1*** as well.

Theorem 8. *Provided P is **R3**, and Q is **R1** and **R3**,*

$$\boldsymbol{R3}(P\ ;\ Q) = P\ ;\ Q \qquad \textit{R3-sequence-closure}$$

This is not a problem because, as detailed in the next section, we actually work with the theory characterised by all healthiness conditions.

As required, ***R3*** is an idempotent.

Law 106 (R3-idempotent). $\boldsymbol{R3} \circ \boldsymbol{R3} = \boldsymbol{R3}$

Since $\mathbb{I}_{rea}$ specifies behaviour for when $\neg\ okay$ holds, it should not be a big surprise that ***R3*** also does not commute with ***H1***. It does commute with the other healthiness conditions, though.

Law 107 (commutativity-*R3*-*H2*). $\boldsymbol{R3} \circ \boldsymbol{H2} = \boldsymbol{H2} \circ \boldsymbol{R3}$

Law 108 (commutativity-*R3*-*R1*). $\boldsymbol{R3} \circ \boldsymbol{R1} = \boldsymbol{R1} \circ \boldsymbol{R3}$

Law 109 (commutativity-*R3*-*R2*). $\boldsymbol{R3} \circ \boldsymbol{R2} = \boldsymbol{R2} \circ \boldsymbol{R3}$

Moreover, if all that there is about a process that is not ***H1*** is the fact that it specifies the behaviour required by ***R1***, then we have a commutativity property. This sort of property is important because we are going to express reactive processes as reactive designs.

*Example 14 (**R3**-**H1**-non-commutativity).* Why do ***R3*** and ***H1*** not commute?

$$\begin{array}{lr}
\quad \boldsymbol{H1} \circ \boldsymbol{R3}(P) & \boldsymbol{H1},\ \boldsymbol{R3} \\
= okay \Rightarrow (\mathbb{I}_{rea} \lhd wait \rhd P) & \odot\text{-conditional} \\
= (okay \Rightarrow \mathbb{I}_{rea}) \lhd wait \rhd (okay \Rightarrow P) & \boldsymbol{H1} \\
= \boldsymbol{H1}(\mathbb{I}_{rea}) \lhd wait \rhd \boldsymbol{H1}(P) & \mathbb{I}_{rea} \text{ is not } \boldsymbol{H1} \\
\neq \mathbb{I}_{rea} \lhd wait \rhd \boldsymbol{H1}(P) & \boldsymbol{R3} \\
= \boldsymbol{R3} \circ \boldsymbol{H1}(P) &
\end{array}$$

This derivation shows the precise reason: it is because $\mathbb{I}_{rea}$ is not ***H1***.

This last example explains the need for the weaker commutativity laws below.

Law 110 (*R3*-*H1* sub-commutativity). $\boldsymbol{H1} \circ \boldsymbol{R3} = \boldsymbol{H1} \circ \boldsymbol{R3} \circ \boldsymbol{H1}$

Law 111 (*R3*-*H1*-*R1* sub-commutativity).

$$\boldsymbol{R3} \circ \boldsymbol{R1} \circ \boldsymbol{H1} = \boldsymbol{R1} \circ \boldsymbol{H1} \circ \boldsymbol{R3}$$

Just like ***R1*** and ***R2***, ***R3*** is monotonic and so gives us a complete lattice when applied to the space of designs.

5.5 $\boldsymbol{R}$

A reactive process is a relation that includes in its alphabet *okay*, *tr*, *wait*, and *ref*, and their dashed counterparts, and satisfies the three healthiness conditions $\boldsymbol{R1}$, $\boldsymbol{R2}$, and $\boldsymbol{R3}$. We define $\boldsymbol{R}$ as the composition of these three functions.

$$\boldsymbol{R} \mathrel{\widehat{=}} \boldsymbol{R1} \circ \boldsymbol{R2} \circ \boldsymbol{R3}$$

Since each of the healthiness conditions $\boldsymbol{R1}$, $\boldsymbol{R2}$, and $\boldsymbol{R3}$ commute, their order in the definition above is irrelevant.

Reactive processes have a left zero.

Law 112 (reactive-left-zero). $(tr \leq tr') \mathbin{;} P = tr \leq tr'$

Proof. First, we expand $\boldsymbol{R3}(P)$.

$$\begin{array}{lr}
& \boldsymbol{R3}(P) \\
= & \mathbb{I}_{rea} \lhd wait \rhd P & \mathbb{I}_{rea} \\
= & (\mathbb{I}_{rel} \lhd okay \rhd tr \leq tr') \lhd wait \rhd P & \text{conditional} \\
= & (\neg\, okay \wedge wait \wedge tr \leq tr') \vee (okay \wedge wait \wedge \mathbb{I}_{rel}) \vee (\neg\, wait \wedge P)
\end{array}$$

Now we can prove our result.

$$\begin{array}{lr}
& tr \leq tr';\ P & \text{assumption: } P \text{ is } \boldsymbol{R3} \\
= & tr \leq tr';\ \boldsymbol{R3}(P) & \boldsymbol{R3}(P) \text{ expansion} \\
= & tr \leq tr';\ \neg\, okay \wedge wait \wedge tr \leq tr' & \text{right-one-point, } \mathbb{I}_{rel} \\
& \vee\ tr \leq tr';\ okay \wedge wait \wedge \mathbb{I}_{rel} & \\
& \vee\ tr \leq tr';\ \neg\, wait \wedge P & \\
= & (tr \leq tr';\ tr \leq tr') \vee (tr \leq tr') \vee (tr \leq tr';\ P) & \text{sequence} \\
= & (tr \leq tr') \vee (tr \leq tr';\ \wedge P) & \text{assumption: } P \text{ is } \boldsymbol{R1} \\
= & (tr \leq tr') \vee (tr \leq tr';\ \wedge P \wedge tr \leq tr') & \text{sequence transitivity} \\
= & tr \leq tr' \vee ((tr \leq tr';\ \wedge P \wedge tr \leq tr') \wedge tr \leq tr') & \text{absorption} \\
= & tr \leq tr'
\end{array}$$

Reactive processes also have a restricted identity.

Law 113 (reactive-restricted-identity).

$$\mathbb{I}_{rea} \mathbin{;} P \;=\; P \lhd okay \rhd tr \leq tr'$$

Substitution for *wait* cannot distribute through $\boldsymbol{R}$, since it does not distribute through $\boldsymbol{R3}$; however, it does have the expected simplification properties. Finally, substitution for $okay'$ does not quite distribute through $\boldsymbol{R}$, since it interferes with $\mathbb{I}_{rea}$. The following reductions hold for these substitutions.

Law 114 ($\boldsymbol{R}$-*wait*-false). $(\boldsymbol{R}(P))_f = \boldsymbol{R1} \circ \boldsymbol{R2}(P_f)$

Law 115 (*R-wait*-true). $(\boldsymbol{R}(P))_t = (\mathit{II}_{rea})_t$

Law 116 (*R-okay′*). $(\boldsymbol{R}(P))^c = ((\mathit{II}_{rea})^c \lhd \mathit{wait} \rhd \boldsymbol{R1} \circ \boldsymbol{R2}(P^c))$

The set of reactive processes is closed under the program operators.

Theorem 9. *Provided P and Q are $\boldsymbol{R}$-healthy,*

$$\boldsymbol{R}(P \wedge Q) = P \wedge Q \qquad \textit{R-}\wedge\textit{-closure}$$
$$\boldsymbol{R}(P \vee Q) = P \vee Q \qquad \textit{R-}\vee\textit{-closure}$$
$$\boldsymbol{R}(P \lhd tr' = tr \rhd Q) = P \lhd tr' = tr \rhd Q \qquad \textit{R-conditional-closure}$$
$$\boldsymbol{R}(P \,;\, Q) = P \,;\, Q \qquad \textit{R-sequence-closure}$$

Since ***R1***, ***R2***, and ***R3*** are all monotonic, so is their composition, and so the set of reactive processes is a complete lattice. The ***R***-image of any complete lattice is also a complete lattice. In particular, the ***R***-image of the lattice of designs is a complete lattice. This image turns out to be the set of CSP processes, as we establish in the next section.

6 CSP Processes

A CSP process is a reactive process satisfying two other healthiness conditions.

6.1 *CSP1*

The first healthiness condition requires that, in case of divergence, extension of the trace is the only property that is guaranteed.

$$\boldsymbol{CSP1}(P) \;=\; P \;\vee\; \neg\, \mathit{okay} \wedge tr \leq tr'$$

It is important to observe that ***R1*** requires that, in whatever situation, the trace can only be increased. On the other hand, ***CSP1*** states that, if we are in a divergent state, $\neg\ \mathit{okay}$, then there is no other guarantee.

Exercise 14. Give an example of a reactive process that is ***R1***, but not ***CSP1***.

CSP1 is a combination of ***R1*** and ***H1***; however, like ***R1***, ***CSP1*** does not commute with ***H1***. The reason is the same: it specifies behaviour for when $\neg\ \mathit{okay}$ holds. The lack of commutativity means that, when applying ***R1*** and ***H1***, the order is relevant. As a matter of fact, ***CSP1*** determines the order that should be used, for processes that are already ***R1***.

Law 117 (*CSP1-R1-H1*).

$$\boldsymbol{CSP1}(P) = \boldsymbol{R1} \circ \boldsymbol{H1}(P) \textbf{ provided } P \textit{ is } \boldsymbol{R1}\textit{-healthy}$$

As expected, ***CSP1*** is an idempotent.

$$\boldsymbol{CSP1} \circ \boldsymbol{CSP1} = \boldsymbol{CSP1}$$

The usual closure properties hold for ***CSP1*** processes.

Theorem 10. *Provided P and Q are* ***CSP1****-healthy,*

$$\boldsymbol{CSP1}(P \wedge Q) = P \wedge Q \qquad \textit{CSP1-}\wedge\textit{-closure}$$
$$\boldsymbol{CSP1}(P \vee Q) = P \vee Q \qquad \textit{CSP1-}\vee\textit{-closure}$$
$$\boldsymbol{CSP1}(P \lhd c \rhd Q) = P \lhd c \rhd Q \qquad \textit{CSP1-conditional-closure}$$
$$\boldsymbol{CSP1}(P \,;\, Q) = P \,;\, Q \qquad \textit{CSP1-sequence-closure}$$

This new healthiness condition is independent from the previous ones.

Law 118 (commutativity-*CSP1*-*R1*). $\boldsymbol{CSP1} \circ \boldsymbol{R1} = \boldsymbol{R1} \circ \boldsymbol{CSP1}$

Law 119 (commutativity-*CSP1*-*R2*). $\boldsymbol{CSP1} \circ \boldsymbol{R2} = \boldsymbol{R2} \circ \boldsymbol{CSP1}$

Law 120 (commutativity-*CSP1*-*R3*). $\boldsymbol{CSP1} \circ \boldsymbol{R3} = \boldsymbol{R3} \circ \boldsymbol{CSP1}$

A reactive process defined in terms of a design is always ***CSP1***-healthy. This is because the design does not restrict the behaviour when $\neg$ *okay* holds, and ***R*** insists only that $tr \leq tr'$.

Law 121 (reactive-design-*CSP1*). $\boldsymbol{CSP1}(\boldsymbol{R}(P \vdash Q)) = \boldsymbol{R}(P \vdash Q)$

If an ***R1***-healthy predicate R appears in a design's postcondition, in the scope of another predicate that is also ***R1***, then R is ***CSP1***-healthy. This is because, for ***R1*** predicates, ***CSP1*** amounts to the composition of ***H1*** and ***R1***. A similar law applies to the negation of such a ***CSP1*** predicate.

Law 122 (design-post-and-*CSP1*).

$$P \vdash (Q \wedge \boldsymbol{CSP1}(R))) \;=\; (P \vdash Q \wedge R)$$
provided Q and R are ***R1***-healthy

Law 123 (design-post-and-not-*CSP1*).

$$P \vdash (Q \wedge \neg\, \boldsymbol{CSP1}(R))) \;=\; (P \vdash Q \wedge \neg\, R)$$
provided *Q and R are* ***R1****-healthy*

These two laws are combined in the following law that eliminates ***CSP1*** from the condition of a conditional.

Law 124 (design-post-conditional-*CSP1*).

$$(P \vdash (Q \lhd \boldsymbol{CSP1}(R) \rhd S)) \;=\; (P \vdash (Q \lhd R \rhd S))$$
provided *Q, R and S are* ***R1****-healthy*

Proof.

$$
\begin{array}{ll}
& P \vdash (Q \lhd \mathbf{CSP1}(R) \rhd S) \qquad \text{conditional} \\
= & P \vdash (Q \wedge \mathbf{CSP1}(R)) \vee (S \wedge \neg\, \mathbf{CSP1}(R)) \quad \text{design, propositional calculus} \\
= & (P \vdash Q \wedge \mathbf{CSP1}(R)) \vee (P \vdash S \wedge \neg\, \mathbf{CSP1}(R)) \\
& \qquad \text{design-post-and-}\mathbf{CSP1}\text{, assumption: } Q \text{ and } R \text{ are } \mathbf{R1}\text{-healthy} \\
= & (P \vdash Q \wedge R) \vee (P \vdash S \wedge \neg\, \mathbf{CSP1}(R)) \\
& \qquad \text{design-post-and-not-}\mathbf{CSP1}\text{, assumption: } S \text{ is } \mathbf{R1}\text{-healthy} \\
= & (P \vdash Q \wedge R) \vee (P \vdash S \wedge \neg\, R) \quad \text{design, propositional calculus, conditional} \\
= & P \vdash (Q \lhd R \rhd S)
\end{array}
$$

Substitution for *wait* and $okay'$ distributes through $\mathbf{CSP1}$.

Law 125 ($\mathbf{CSP1}$-*wait*-$okay'$).

$$(\mathbf{CSP1}(P))^c_b \;=\; \mathbf{CSP1}(P^c_b) \textbf{ provided } P \textit{ is } \mathbf{R1}\textit{-healthy}$$

The many restrictions on these laws related to $\mathbf{R1}$ healthiness are not a problem, since $\mathbf{CSP1}$ is a healthiness condition on reactive processes.

6.2 *CSP2*

The second healthiness condition for CSP processes, $\mathbf{CSP2}$, is defined in terms of J (which was introduced in Section 4) as follows.

$$\mathbf{CSP2}(P) = P \,;\, J$$

It is a direct consequence of Theorem 3 that $\mathbf{CSP2}$ is a recast of $\mathbf{H2}$, now with an extended alphabet that includes *okay*, *wait*, *tr*, and *ref*. In other words, in the theory of CSP processes, we let go of $\mathbf{H1}$, but we retain $\mathbf{H2}$, under another disguise.

Idempotence and commutative properties for $\mathbf{CSP2}$ follow from those for $\mathbf{H2}$. We add only that it commutes with $\mathbf{CSP1}$.

Law 126 (commutativity-$\mathbf{CSP2}$-$\mathbf{CSP1}$).

$$\mathbf{CSP2} \circ \mathbf{CSP1} = \mathbf{CSP1} \circ \mathbf{CSP2}$$

Closure of designs is not established considering $\mathbf{H1}$ and $\mathbf{H2}$ individually; we consider $\mathbf{H2}$, or $\mathbf{CSP2}$ rather, below. It is not closed with respect to conjunction, and it is not difficult to prove that $P \wedge Q \sqsubseteq \mathbf{CSP2}(P \wedge Q)$, providing P and Q are $\mathbf{CSP2}$.

Theorem 11. *Provided P and Q are $\mathbf{CSP2}$-healthy,*

$$\mathbf{CSP2}(P \vee Q) = P \vee Q \qquad \mathbf{CSP2}\text{-}\vee\text{-closure}$$

$$\mathbf{CSP2}(P \lhd b \rhd Q) = P \lhd b \rhd Q \qquad \mathbf{CSP2}\text{-conditional-closure}$$

$$\mathbf{CSP2}(P \,;\, Q) = P \,;\, Q \qquad \mathbf{CSP2}\text{-sequence-closure}$$

Exercise 15. Prove algebraically that

$$P \wedge Q \sqsubseteq \textbf{\textit{CSP2}}(P \wedge Q)$$

providing P and Q are ***CSP2***-healthy.

Substitution of *true* for $okay'$ does not distribute through ***CSP2***, but produces the disjunction of two cases.

Law 127 (*CSP2*-converge).

$$(\textbf{\textit{CSP2}}(P))^t = P^t \vee P^f$$

Proof.

$$\begin{array}{lr}
& (\textbf{\textit{CSP2}}(P))^t & \textbf{\textit{CSP2}} \\
= & (P \,;\, J)^t & \text{substitution} \\
= & P \,;\, J^t & J \\
= & P \,;\, ((okay \Rightarrow okay') \wedge \mathit{II}_{rel}^{-okay})^t & \text{substitution} \\
= & P \,;\, ((okay \Rightarrow \textbf{true}) \wedge \mathit{II}_{rel}^{-okay}) & \text{propositional calculus} \\
= & P \,;\, \mathit{II}_{rel}^{-okay} & \text{propositional calculus} \\
= & P \,;\, (okay \vee \neg\, okay) \wedge \mathit{II}_{rel}^{-okay} & \text{relational calculus} \\
= & P \,;\, okay \wedge \mathit{II}_{rel}^{-okay} \vee P \,;\, \neg\, okay \wedge \mathit{II}_{rel}^{-okay} & okay\text{-boolean, } \mathit{II}_{rel}^{-okay} \\
= & P \,;\, okay = true \wedge tr = tr' \wedge ref = ref' \wedge wait = wait' & \text{one-point} \\
& \vee \\
& P \,;\, okay = false \wedge tr = tr' \wedge ref = ref' \wedge wait = wait' & \\
= & P^t \vee P^f &
\end{array}$$

Substitution of *false* for $okay'$ eliminates ***CSP2***.

Law 128 (*CSP2*-diverge).

$$(\textbf{\textit{CSP2}}(P))^f = P^f$$

Proof.

$$\begin{array}{lr}
& (\textbf{\textit{CSP2}}(P))^f & \textbf{\textit{CSP2}} \\
= & (P \,;\, J)^f & \text{substitution} \\
= & P \,;\, J^f & J \\
= & P \,;\, ((okay \Rightarrow okay') \wedge \mathit{II}_{rel}^{-okay})^f & \text{substitution} \\
= & P \,;\, ((okay \Rightarrow \textbf{false}) \wedge \mathit{II}_{rel}^{-okay}) & \text{propositional calculus} \\
= & P \,;\, (\neg\, okay \wedge \mathit{II}_{rel}^{-okay}) & okay\text{-boolean, } \mathit{II}_{rel}^{-okay} \\
= & P \,;\, (okay = false \wedge tr = tr' \wedge ref = ref' \wedge wait = wait') & \text{one-point} \\
= & P^f &
\end{array}$$

It is trivial to prove that any reactive design is ***CSP2***, since ***CSP2*** and ***H2*** are the same. A reactive process defined in terms of a design is always ***CSP2***-healthy.

Law 129 (reactive-design-*CSP2*). $\boldsymbol{CSP2}(\boldsymbol{R}(P \vdash Q)) = \boldsymbol{R}(P \vdash Q)$

A CSP process is a reactive process that is both ***CSP1*** and ***CSP2***-healthy. The following theorem shows that any CSP process can be specified in terms of a design using $\boldsymbol{R}$.

Theorem 12. *For every CSP process P,* $P = \boldsymbol{R}(\neg\, P_f^f \vdash P_f^t)$

Together with Laws 121 and 129, this theorem accounts for a style of specification for CSP processes in which we use a design to give its behaviour when the previous process has terminated and not diverged, and leave the definition of the behaviour in the other situations for the healthiness conditions. The precondition of the design characterises the conditions that guarantee that the process does not diverge: it is not the case that, having started (*wait* is *false*), then it diverges (*okay′* is *false*). The postcondition gives the behaviour when, having started, the process does not diverge (*okay′* is *true*).

Figure 1 summarises the relationship between the theories of the UTP we presented so far. In black, we have all the alphabetised predicates; in white, we have the relations: those predicates whose alphabet include only dashed and undashed variables. Designs and reactive processes are disjoint sets of relations. Finally, CSP processes are reactive; moreover, they are the $\boldsymbol{R}$-image of designs.

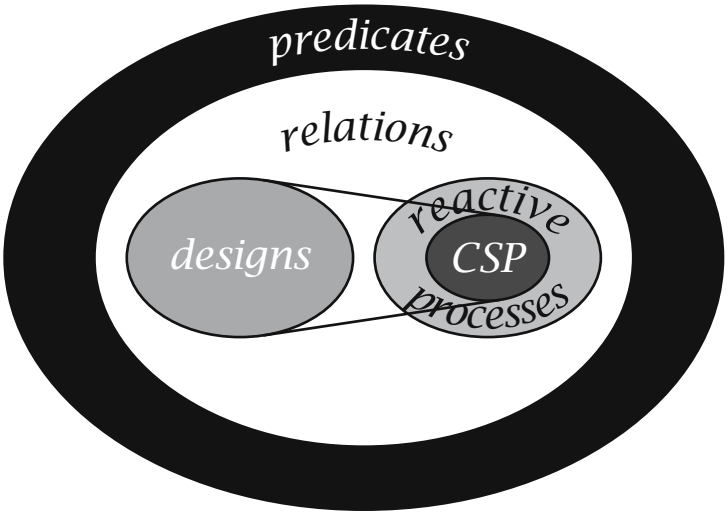


Fig. 1. UTP theories

Motivated by the result above, we express some constructs of CSP as reactive designs. We show that our definitions are the same as those in [117], with a few

exceptions that we explain. Before we proceed, however, we observe that, for CSP processes, $\mathbb{I}_{rea}$ is an identity.

Law 130 ($\mathbb{I}_{rea}$-sequence-CSP).

$$\mathbb{I}_{rea} \,;\, P = P \textbf{ provided } P \textit{ is both } \textbf{\textit{CSP1}} \textit{ and } \textbf{\textit{CSP2}}\textit{-healthy}$$

In spite of its name, $\mathbb{I}_{rea}$ is not a true identity for reactive processes that are not CSP.

Exercise 16. Give an example of a reactive process P for which $\mathbb{I}_{rea};\ P \neq P$.

6.3 *STOP*

We want the following definition for *STOP*.

$$STOP = \mathbf{R}(\mathbf{true} \vdash tr' = tr \wedge wait')$$

Since *STOP* deadlocks, it does not change the trace or terminates. Moreover, all events can be refused; so, we leave the value of ref' unrestrained: any refusal set is a valid observation.

The next law describes the effect of starting *STOP* properly and insisting that it does not diverge. The result is that it leaves the trace unchanged and it waits forever. We need to apply ***CSP1***, since we have not ruled out the possibility of its predecessor diverging.

Law 131 (*STOP*-converge). $STOP^t_f = \textbf{\textit{CSP1}}(tr' = tr \wedge wait')$

Proof.

$$\begin{array}{lr}
STOP^t_f & STOP \\
= (\mathbf{R}(\mathbf{true} \vdash tr' = tr \wedge wait'))^t_f & \textbf{\textit{R}}\textit{-wait-false}, \textbf{\textit{R1}}\textit{-okay}', \textbf{\textit{R2}}\textit{-okay}' \\
= \textbf{\textit{R1}} \circ \textbf{\textit{R2}}((\mathbf{true} \vdash tr' = tr \wedge wait')^t_f) & \text{substitution} \\
= \textbf{\textit{R1}} \circ \textbf{\textit{R2}}((\mathbf{true} \vdash tr' = tr \wedge wait')^t) & \text{design, substitution} \\
= \textbf{\textit{R1}} \circ \textbf{\textit{R2}}(okay \Rightarrow tr' = tr \wedge wait') & \textbf{\textit{R2}} \\
= \textbf{\textit{R1}}(okay \Rightarrow tr' = tr \wedge wait') & \textbf{\textit{H1}} \\
= \textbf{\textit{R1}}(\textbf{\textit{H1}}(tr' = tr \wedge wait')) & \textbf{\textit{R1}} \\
= \textbf{\textit{R1}}(\textbf{\textit{H1}}(\textbf{\textit{R1}}(tr' = tr \wedge wait'))) & \textbf{\textit{CSP1}}\textit{-}\textbf{\textit{R1}}\textit{-}\textbf{\textit{H1}} \text{ and } \textbf{\textit{R1}} \\
= \textbf{\textit{CSP1}}(tr' = tr \wedge wait') &
\end{array}$$

Now we consider the behaviour if we start *STOP* properly, but insist that it does diverge. Of course, *STOP* cannot do this, so the result is that it could not have been started.

Law 132 (*STOP*-diverge). $STOP^f_f = \textbf{\textit{R1}}(\neg\ okay)$

Proof.

$$
\begin{array}{lr}
STOP_f^f & STOP \\
= (\boldsymbol{R}(\mathbf{true} \vdash tr' = tr \wedge wait'))_f^f & \boldsymbol{R}\text{-}wait\text{-}false,\ \boldsymbol{R1}\text{-}okay',\ \boldsymbol{R2}\text{-}okay' \\
= \boldsymbol{R1} \circ \boldsymbol{R2}((\mathbf{true} \vdash tr' = tr \wedge wait')_f^f) & \text{substitution} \\
= \boldsymbol{R1} \circ \boldsymbol{R2}((\mathbf{true} \vdash tr' = tr \wedge wait')^f) & \text{design, substitution} \\
= \boldsymbol{R1} \circ \boldsymbol{R2}(\neg\ okay) & \boldsymbol{R2} \\
= \boldsymbol{R1}(\neg\ okay) &
\end{array}
$$

It is possible to prove the following law for *STOP*: it is a left zero for sequence.

Law 133 (STOP-left-zero). $STOP\ ;\ P = STOP$

This gives some reassurance of the validity of our definition.

6.4 *SKIP*

In the UTP, the definition of *SKIP* is as follows.

$$SKIP \mathrel{\widehat{=}} \boldsymbol{R}(\exists\ ref \bullet \Pi_{rea})$$

We propose the formulation presented in the law below.

Law 134 (SKIP-reactive-design). $SKIP = \boldsymbol{R}(\mathbf{true} \vdash tr' = tr \wedge \neg\ wait')$

This characterises *SKIP* as the program that terminates immediately without changing the trace; the refusal set is left unspecified, as it is irrelevant after termination.

6.5 *CHAOS*

The UTP definition for *CHAOS* is $\boldsymbol{R}(\mathbf{true})$. Instead of **true**, we use a design.

Law 135 (CHAOS-reactive-design). $CHAOS = \boldsymbol{R}(\mathbf{false} \vdash \mathbf{true})$

It is perhaps not surprising that *CHAOS* is the reactive abort.

Law 136 (CHAOS-left-zero). $CHAOS\ ;\ P = CHAOS$

The new characterisation of *CHAOS* can be used in the proof of the law above.

6.6 External Choice

For CSP processes P and Q with a common alphabet, their external choice is defined as follows.

$$P \,\Box\, Q \mathrel{\widehat{=}} \boldsymbol{CSP2}((P \wedge Q) \lhd STOP \rhd (P \vee Q))$$

This says that the external choice behaves like the conjunction of P and Q if no progress has been made (that is, if no event has been observed and termination has not occurred). Otherwise, it behaves like their disjunction. This is an economical definition, and we believe that its re-expression as a reactive design is insightful. To prove the law that gives this description, we need a few lemmas, which we present below.

In order to present external choice as a reactive design, we need to calculate a meaningful description for the design $\mathbf{R}(\neg\ (P\ \Box\ Q)^f_f \vdash (P\ \Box\ Q)^t_f)$. that is indicated by Theorem 12. We start with the precondition, and calculate a result for $(P\ \Box\ Q)^f_f$.

Lemma 15 (external-choice-diverge). *Provided P and Q are **R1**-healthy,* $(P\ \Box\ Q)^f_f = (P^f_f \vee Q^f_f) \lhd okay \rhd (P^f_f \wedge Q^f_f)$

This result needs to be negated, but it remains a conditional on the value of *okay*. Since it is a precondition, this conditional may be simplified.

Lemma 16 (external-choice-precondition).

$$(\neg\ (P\ \Box\ Q)^f_f \vdash R\ )\ =\ (\neg\ (P^f_f \vee Q^f_f) \vdash R\ )$$

Now we turn our attention to the postcondition.

Lemma 17 (external-choice-converge).

$$(P\ \Box\ Q)^t_f\ =$$
$$(P \wedge Q) \lhd STOP \rhd (P \vee Q)^t_f \vee (P \wedge Q) \lhd STOP \rhd (P \vee Q)^f_f$$

The second part of the postcondition is in contradiction with the precondition, and when we bring the two together it can be removed. The conditional on *STOP* can then be simplified.

Lemma 18 (design-external-choice-lemma).

$$(\neg\ (P\ \Box\ Q)^f_f \vdash (P\ \Box\ Q)^t_f)\ =$$
$$((\neg\ P^f_f \wedge Q^f_f) \vdash ((P^t_f \wedge Q^t_f) \lhd tr' = tr \wedge wait' \rhd (P^t_f \vee Q^t_f)))$$

Finally, we collect our results to give external choice as a reactive design.

Law 137 (design-external-choice).

$$P\ \Box\ Q\ =\ \mathbf{R}((\neg\ P^t_f \wedge \neg\ Q^t_f) \vdash (P^t_f \wedge Q^t_f) \lhd tr' = tr \wedge wait' \rhd (P^t_f \vee Q^t_f))$$

Proof.

$$\begin{array}{ll} & P\ \Box\ Q & \text{CSP-reactive-design} \\ = & \mathbf{R}(\neg\ (P\ \Box\ Q)^f_f \vdash (P\ \Box\ Q)^t_f) & \text{design-external-choice-lemma} \\ = & \mathbf{R}((\neg\ P^f_f \wedge \neg\ Q^f_f) \vdash (P^t_f \wedge Q^t_f \lhd tr' = tr \wedge wait' \rhd P^t_f \vee Q^t_f)) & \end{array}$$

The design in this law describes the behaviour of an external choice $P \Box Q$ when its predecessor has terminated without diverging. In this case, the external choice does not diverge if neither P nor Q does; this is captured in the precondition. The postcondition establishes that if there has been no activity, or rather, the trace has not changed and the choice has not terminated, then the behaviour is given by the conjunction of P and Q. If there has been any activity, then the choice has been made and the behaviour is either that of P or that of Q.

Exercise 17. Write out the reactive design that corresponds to the external choice below, where a and b are events.

$$\begin{array}{l} \boldsymbol{R}(true \vdash wait' \wedge tr' = tr \wedge \{\, a \,\} \notin ref' \vee \neg\, wait' \wedge tr' = tr \frown \langle\, a \,\rangle) \\ \Box \\ \boldsymbol{R}(true \vdash wait' \wedge tr' = tr \wedge \{\, b \,\} \notin ref' \vee \neg\, wait' \wedge tr' = tr \frown \langle\, a \,\rangle) \end{array}$$

Exercise 18. How can we write the process below in the notation of CSP?

$$\boldsymbol{R}(true \vdash wait' \wedge tr' = tr \wedge \{\, a \,\} \notin ref' \vee \neg\, wait' \wedge tr' = tr \frown \langle\, a \,\rangle)$$

6.7 Extra Healthiness Conditions: CSP3 and CSP4

The healthiness conditions ***CSP1*** and ***CSP2*** are not strong enough to characterise a UTP model containing only those relations that correspond to processes that can be written using the CSP operators as presented, for example, in Chapter 3. In principle, we need more healthiness conditions to further restrict the subset of reactive processes of interest. As a matter of fact, however, there are advantages to this greater flexibility. In any case, a few other healthiness conditions can be very useful, if not essential. Here, we present two of these.

CSP3. This healthiness condition requires that the behaviour of a process does not depend on the initial value of ref. In other words, it should be the case that, when a process P starts, whatever the previous process could or could not refuse when it finished should be irrelevant. Formally, the ***CSP3*** healthiness condition is $\neg\, wait \Rightarrow (P = \exists\, ref \bullet P)$. If the previous process diverged, $\neg\, okay$, then ***CSP1*** guarantees that the behaviour of P is already independent of ref. So, the restriction imposed by ***CSP3*** is really relevant for the situation $okay \wedge \neg\, wait$, as should be expected.

We can express ***CSP3*** in terms of an idempotent defined as follows.

$$\boldsymbol{CSP3}(P) = SKIP \,;\, P$$

The following lemma establishes that this is the right idempotent.

Lemma 19. *P is **CSP3**-healthy if, and only if, $SKIP \,;\, P = P$.*

Using this idempotent, we can prove that $SKIP$ is ***CSP3***-healthy.

Law 138 (SKIP-*CSP3*). $\boldsymbol{CSP3}(SKIP) = SKIP$

With this result, it is very simple to prove that ***CSP3*** is indeed an idempotent.

$$\boldsymbol{CSP3} \circ \boldsymbol{CSP3} = \boldsymbol{CSP3}$$

Since CSP processes are not closed with respect to conjunction, we only worry about closure of the extra healthiness conditions with respect to the other programming operators.

Theorem 13. *Provided P and Q are* ***CSP3****-healthy,*

$$\boldsymbol{CSP3}(P \vee Q) = P \vee Q \qquad \textbf{CSP3-}\vee\textbf{-closure}$$
$$\boldsymbol{CSP3}(P \lhd tr = tr' \rhd Q) = P \lhd tr = tr' \rhd Q \qquad \textbf{CSP3-conditional-closure}$$
$$\boldsymbol{CSP3}(P \mathbin{;} Q) = P \mathbin{;} Q \qquad \textbf{CSP3-sequence-closure}$$

CSP4. The second extra healthiness condition, ***CSP4***, is similar to ***CSP3***.

$$\boldsymbol{CSP4}(P) = P \mathbin{;} SKIP$$

It requires that, on termination or divergence, the value of ref' is irrelevant. The following lemma makes this clear.

Lemma 20.

$$\begin{aligned} P \mathbin{;} SKIP = {} & (\exists\, ref' \bullet P) \wedge okay' \wedge \neg\, wait' \vee \\ & P \wedge okay' \wedge wait' \vee \\ & (P \wedge \neg\, okay') \mathbin{;} tr \leq tr' \end{aligned}$$

This result shows that, if $P = P \mathbin{;} SKIP$, then if P has terminated without diverging ($okay' \wedge \neg\, wait'$), the value of ref' is not relevant. If P has not terminated ($okay' \wedge wait'$), then the value of ref' is as defined by P itself. Finally, if it diverges ($\not okay'$), then the only guarantee is that the trace is extended; the value of the other variables is irrelevant.

It is easy to prove that $SKIP$, $STOP$, and $CHAOS$ are ***CSP4***-healthy.

Law 139 (SKIP-*CSP4*). $\boldsymbol{CSP4}(SKIP) = SKIP$

Law 140 (STOP-*CSP4*). $\boldsymbol{CSP4}(STOP) = STOP$

Law 141 (CHAOS-*CSP4*). $\boldsymbol{CSP4}(CHAOS) = CHAOS$

The usual closure properties also hold.

Theorem 14. *Provided P and Q are* ***CSP4****-healthy,*

$$\boldsymbol{CSP4}(P \vee Q) = P \vee Q \qquad \textbf{CSP4-}\vee\textbf{-closure}$$
$$\boldsymbol{CSP4}(P \lhd b \rhd Q) = P \lhd b \rhd Q \qquad \textbf{CSP4-conditional-closure}$$
$$\boldsymbol{CSP4}(P \mathbin{;} Q) = P \mathbin{;} Q \qquad \textbf{CSP4-sequence-closure}$$

As detailed in the next section, other healthiness conditions may be useful. We leave this search as future work; [117] presents an additional healthiness condition that we omit here: ***CSP5***.

7 Failures-Divergences Model

The failures-divergences model is the definitive reference for the semantics of CSP [225]. It is formed by a set F of pairs and a set D of traces. The pairs are the failures of the process. A failure is formed by a trace and a set of events; the trace s records a possible history of interaction, and the set includes the events that the process may refuse after the interactions in the trace. This set is the refusals of P after s. The set D of traces is the divergences of the process. After engaging in the interactions in any of these traces, the process may diverge.

Refinement in this model is defined as reverse containment. A process P_1 is refined by a process P_2 if, and only if, the set of failures and the set of divergences of P_2 are contained or equal to those of P_1.

The simpler traces model includes only a set of traces. For a process P, the set $traces_\perp(P)$ contains the set of all traces in which P can engage, including those that lead to or arise from divergence.

7.1 Failures-Divergences Healthiness Conditions

A number of healthiness conditions are imposed on the failures-divergences model. The first healthiness condition requires that the set of traces of a process is captured in its set of failures, that this set is non-empty and prefix closed. This is because the empty trace is a trace of every process, and every earlier record of interaction is a possible interaction of the process.

F1 $\quad traces_\perp(P) = \{\, t \mid (t, X) \in F \,\}$ is non-empty and prefix closed

The next healthiness condition requires that if (s, X) is a failure, then (s, Y) is also a failure, for all subsets Y of X. This means that, if after s the process may refuse all the events of X, then it may refuse all the events in the subsets of X.

$$\textbf{\textit{F2}} \quad (s, X) \in F \land Y \subseteq X \Rightarrow (s, Y) \in F$$

Also concerning refusals, we have a healthiness condition that requires that if an event is not possible, according to the set of traces of the process, then it must be in the set of refusals.

$$\textbf{\textit{F3}} \quad (s, X) \in F \land (\forall\, a : Y \bullet s \frown \langle\, a \,\rangle \notin traces_\perp(P)) \Rightarrow (s, X \cup Y) \in F$$

The event ✓ is used to mark termination. The following healthiness condition requires that, just before termination, a process can refuse all interactions. The set Σ includes all the events in which the process can engage, except ✓ itself.

$$\textbf{\textit{F4}} \quad s \frown \langle\, ✓ \,\rangle \in traces_\perp(P) \Rightarrow (s, \Sigma) \in F$$

The last three healthiness conditions are related to the divergences of a process. First, if a process can diverge after engaging in the events of a trace s, then it

can diverge after engaging in the events of any extension of s. The idea is that, conceptually, after divergence, any behaviour is possible. Even ✓ is included in the extended traces, and not necessarily as a final event. The set Σ^* includes all traces on events in Σ, and $\Sigma^{*\checkmark}$ includes all traces on events in $\Sigma \cup \{ \checkmark \}$.

D1 $\quad s \in D \cap \Sigma^* \land t \in \Sigma^{*\checkmark} \Rightarrow s \frown t \in D$

The next condition requires that, after divergence, all events may be refused.

D2 $\quad s \in D \Rightarrow (s, X) \in F$

The final healthiness condition requires that if a trace that marks a termination is in the set of divergences, it is because the process diverged before termination. It would not make sense to say that a process diverged after it terminated.

D3 $\quad s \frown \langle \checkmark \rangle \in D \Rightarrow s \in D$

Some of these healthiness conditions correspond to UTP healthiness conditions. Some of them are not contemplated. They are discussed individually later on.

7.2 Failures-Divergences Model of a UTP Process

We can calculate a failures-divergences representation of a UTP process. More precisely, we define a few functions that take a UTP predicate and return a component of the failures-divergences model. We first define a function *traces*; it takes a UTP predicate P and returns the set of traces of the corresponding process.

In the UTP model, the behaviour of a process is that prescribed when *okay* and $\neg$ *wait*. The behaviour in the other cases is determined by the UTP healthiness conditions, and is included in the UTP model so that sequence is simplified: it is just relational composition. In the failures-divergences model, this extra behaviour is not captured and is enforced in the definition of sequence.

The value of tr records the history of events before the start of the process; tr' carries this history forward. This simplifies the definition of sequence. In the failures-divergences model, this extra behaviour is not captured. Therefore, the traces in the set $traces(P)$ are the sequences $tr' - tr$ that arise from the behaviour of P itself.

$$\begin{aligned} traces(P) = \{\, tr' - tr \mid okay \land \neg\, wait \land P \land okay' \,\} \cup \\ \{\, (tr' - tr) \frown \langle \checkmark \rangle \mid okay \land \neg\, wait \land P \land okay' \land \neg\, wait' \,\} \end{aligned}$$

The set $traces(P)$ only includes the traces that lead to non-divergent behaviour. Moreover, if a trace $tr' - tr$ leads to termination, $\neg\, wait'$, then $traces(P)$ also includes $(tr' - tr) \frown \langle \checkmark \rangle$, since ✓ is used in the failures-divergences model to signal termination.

Exercise 19. Calculate $traces(STOP)$.

The traces that lead to or arise from divergent behaviour are those in the set $divergences(P)$ defined below.

$$divergences(P) = \{\, tr' - tr \mid okay \wedge \neg\, wait \wedge P \wedge \neg\, okay' \,\}$$

Exercise 20. Calculate $divergences(STOP)$.

The set $traces_\perp(P)$ mentioned in the healthiness conditions of the failures-divergences model includes both the divergent and non-divergent traces.

$$traces_\perp(P) = traces(P) \cup divergences(P)$$

The failures are recorded for those states that are stable (non-divergent) or final.

$$\begin{array}{l} failures(P) = \\ \quad \{\, ((tr' - tr), ref') \mid okay \wedge \neg\, wait \wedge P \wedge okay' \,\} \cup \\ \quad \{\, ((tr' - tr) \frown \langle \checkmark \rangle, ref') \mid okay \wedge \neg\, wait \wedge P \wedge okay' \wedge \neg\, wait' \,\} \cup \\ \quad \{\, ((tr' - tr) \frown \langle \checkmark \rangle, ref' \cup \{ \checkmark \}) \mid okay \wedge \neg\, wait \wedge P \wedge okay' \wedge \neg\, wait' \,\} \end{array}$$

For the final state, the extra trace $(tr' - tr) \frown \langle \checkmark \rangle$ is recorded. Also, after termination, for every refusal set ref', there is an extra refusal set $ref' \cup \{ \checkmark \}$. This is needed because $\checkmark$ is not part of the UTP model and is not considered in the definition of ref'.

Exercise 21. Calculate $failures(STOP)$ and $failures(SKIP)$.

The set of failures in the failures-divergences model includes failures for the divergent traces as well.

$$failures_\perp(P) = failures(P) \cup \{\, (s, ref) \mid s \in divergences(P) \,\}$$

For a divergent trace, there is a failure for each possible refusal set.

The functions $failures_\perp$ and $divergences$ map the UTP model to the failures-divergences model. In studying the relationship between alternative models for a language, it is usual to hope for an isomorphism between them. In our case, this would amount to finding inverses for $failures_\perp$ and $divergences$. Actually, this is not possible; UTP and the failures-divergences model are not isomorphic. This is discussed in detail below.

7.3 Relationship Between the Failures-Divergences and the UTP Model

The UTP model contains processes that cannot be represented in the failures-divergences model. Some of them are useful in a model for a language that has a richer set of constructions to specify data operations. Others may need to be ruled out by further healthiness conditions.

The failures-divergences model, for example, does not have a top element; all divergence-free deterministic processes are maximal. In the UTP model, $\boldsymbol{R}(\mathbf{true} \vdash \mathbf{false})$ is the top.

Lemma 21. *For every CSP process P, we have that $P \sqsubseteq \boldsymbol{R}(\mathbf{true} \vdash \mathbf{false})$.*

The process $\boldsymbol{R}(\mathbf{true} \vdash \mathbf{false})$ is $(\mathbb{I}_{rea} \lhd wait \rhd \neg okay \land tr \leq tr')$. Its behaviour when *okay* and $\neg$ *wait* is **false**. As such, it is mapped to the empty set of failures and divergences; in other words, it is mapped to *STOP*. Operationally, this can make sense, but *STOP* does not have the same properties of $\boldsymbol{R}(\mathbf{true} \vdash \mathbf{false})$. In particular, it does not refine every other process.

Exercise 22. Give an algebraic proof that

$$\boldsymbol{R}(\mathbf{true} \vdash \mathbf{false}) = (\mathbb{I}_{rea} \lhd wait \rhd \neg okay \land tr \leq tr')$$

Exercise 23. Take advantage of the result of the previous exercise to calculate $failures_{\bot}(\boldsymbol{R}(\mathbf{true} \vdash \mathbf{false}))$ and $divergences(\boldsymbol{R}(\mathbf{true} \vdash \mathbf{false}))$.

Exercise 24. Explain why *STOP* does not refine every other process. Consider both the UTP and the failures-divergences models.

In general terms, every process that behaves miraculously in any of its initial states cannot be accurately represented using a failures-divergences model. We do not, however, necessarily want to rule out such processes, as they can be useful as a model for a state-rich CSP.

If we analyse the range of $failures_{\bot}$ and *divergences*, we can see that it does not satisfy a few of the healthiness conditions ***F1-4*** and ***D1-3***.

F1. The set $traces_{\bot}(P)$ is empty for $P = \boldsymbol{R}(\mathbf{true} \vdash \mathbf{false})$; as discussed above, this can be seen as an advantage. Also, $traces_{\bot}(P)$ is not necessarily prefix closed. For example, the process $\boldsymbol{R}(\mathbf{true} \vdash tr' = tr \frown \langle\, a, b \,\rangle \land \neg\, wait')$ engages in the events a and b and then terminates. It does not have a stable state in which a took place, but b is yet to happen.

Exercise 25. Calculate $traces(\boldsymbol{R}(\mathbf{true} \vdash tr' = tr \frown \langle\, a, b \,\rangle \land \neg\, wait'))$. Prove that $\boldsymbol{R}(\mathbf{true} \vdash tr' = tr \frown \langle\, a, b \,\rangle \land \neg\, wait')$ is both ***CSP3*** and ***CSP4***.

F2. This is also not enforced for UTP processes. It is expected to be a consequence of a healthiness condition ***CSP5*** presented in [117].

F3. Again, it is simple to provide a counterexample.

$$\boldsymbol{R}(\mathbf{true} \vdash tr' = tr \frown \langle\, a \,\rangle \land ref' \subseteq \{\, b \,\} \land wait' \lor tr' = tr \frown \langle\, a, b \,\rangle \land \neg\, wait')$$

In this case, a is not an event that can take place again after it has already occurred, and yet it is not being refused.

Exercise 26. Calculate the failures of the process above, and check that it is both ***CSP3*** and ***CSP4***.

F4. This holds for ***CSP4***-healthy processes.

Theorem 15. *Provided P is **CSP4**-healthy,*

$$s \frown \langle \checkmark \rangle \in traces_{\perp}(P) \Rightarrow (s, \Sigma) \in failures(P)$$

D1. Again, ***CSP4*** is required to ensure ***D1***-healthy divergences.

Theorem 16. *Provided P is **CSP4**-healthy,*

$$s \in divergences(P) \cap \Sigma^* \wedge t \in \Sigma^{*\checkmark} \Rightarrow s \frown t \in divergences(P)$$

D2. This is enforced in the definition of $failures_{\perp}$.

D3. Again, this is a simple consequence of the definition (of *divergences*).

Theorem 17.

$$s \frown \langle \checkmark \rangle \in divergences(P) \Rightarrow s \in divergences(P)$$

We view the definition of extra healthiness conditions on UTP processes to ensure ***F1*** and ***F3*** as a challenging task.

8 Conclusions

We have presented two UTP theories of programming: one for pre-post specifications (designs), and one for reactive processes. They have been brought together to form a theory of CSP processes. This is the starting point for the unification of the two theories, whose logical conclusion is a theory of state-rich CSP processes. This is the basis for the semantics of a new notation called *Circus* [255, 51], which combines Z and CSP.

The theory of designs was only briefly discussed. It is the subject of a companion tutorial [256], where through a series of examples, we have presented the alphabetised relational calculus and its sub-theory of designs. In that paper, we have presented the formalisation of four different techniques for reasoning about program correctness.

Even though this is a tutorial introduction to part of the contents of [117], it contains many novel laws and proofs. Notably, the recasting of external choice as a reactive design can be illuminating. Also, the relationship with the failures-divergences model is original.

We hope to have given a didactic and accessible account of the CSP model in the unifying theories of programming. We have left out, however, the definition of many CSP constructs as reactive designs and the exploration of further healthiness conditions. These are going to be the subject of further work.

In [217], UTP is also used to give a semantics to an integration of Z and CSP, which also includes object-oriented features. In [240], the UTP is extended with constructs to capture real-time properties as a first step towards a semantic model for a timed version of *Circus*. In [83], a theory of general correctness is

characterised as an alternative to designs; instead of ***H1*** and ***H2***, a different healthiness condition is adopted to restrict general relations.

Currently, we are collaborating with colleagues to extend UTP to capture mobility, synchronicity, pointers, and object orientation. In particular, in [52] we propose a UTP model for an object-oriented extension of *Circus* based on the language and results discussed in Chapter 2; the details of that model are part of our ongoing work. As explained in Chapter 2, this model can be used to prove the laws proposed there. We hope to contribute to the development of a theory that can support all the major concepts available in modern programming languages.

Using the Compliance Notation in Industry

Phil Clayton and Colin O'Halloran

Systems Assurance Group
QinetiQ Malvern Technology Centre
Malvern, UK

1 Introduction

Nancy Leveson has observed that few safety failures are due to coding errors [152]; for this reason, it is claimed that verification, although desirable, is not the most cost effective use of a limited budget. Evidence does show that safety failures tend to arise instead from requirements or design decisions [205]; however, low-level implementation decisions can also have a large impact on higher level decisions. For example, the removal of a defensive conditional clause from the source code of the inertial reference system of Ariane 5 would have been safe, except for the requirement to execute the ground-based function during flight [88]. When assessing the safety impact of requirement and design decisions there are always worries about the accuracy of the documentation and whether some decisions have not been recorded, or left implicit.

The software source code is the most accurate and accessible record of all implementation decisions, but it is not easily understood by a human. A means of abstracting the source code to a level where a person, with machine support, can sensibly make judgements about conflicting requirements is required. Abstraction addresses the customer's duty of care to understand what it is they are accepting and maintain the safety of the system during its evolution, a potentially very expensive issue.

We have designed a notation for demonstrating compliance between software and a specification [236, 207], and produced a tool to support the demonstration of compliance [206]. One of the design objectives was to support retrospective formalisation of the development of code from a specification. Initial laboratory use indicated that the tool is best suited to reasoning about the safety properties of code. Software that has been written with safety constraints in mind can be abstracted to formally specified safety properties, which can then be reasoned about and assessed against system-level hazards.

This chapter describes a Compliance Notation [236] for the Z notation [257] and an Ada subset [21]; the notation has been used to perform formal refinement for industrial applications. The chapter starts with an introduction to the Compliance Notation and the tool that processes it. An extended example of a compliance argument that contains formal refinement is given, and then experiences of its use are related. The chapter then goes on to present a specialized application of refinement using the Compliance Notation to demonstrate the correct implementation of control laws that govern control systems. Finally, the role of refinement within a system dependability or safety argument is presented.

A. Cavalcanti, A. Sampaio, and J. Woodcock (Eds.): PSSE 2004, LNCS 3167, pp. 269–314, 2006.

2 A Compliance Notation and Tool

2.1 Overview

The Compliance Notation [236] is intended for use with an Ada compiler and existing Z tools. The Ada compiler carries out static checks on the program. For example, in order for refinement to be carried out, it is necessary to translate Ada types to Z. The Compliance Notation translates all discrete types to Z's integers. As refinement is carried out, an Ada compiler will enforce the Ada type rules on the program; this means that refinement and verification conditions (VCs) need only focus on the logical correctness of the program. A Z type checker is applied to the Z parts of the compliance argument and the Z document. The Compliance Notation has four ingredients: the SPARK [21] subset of Ada, together with Z [257], Knuth's literate programming technique [138], and refinement [192, 257].

2.2 The Specification Notation Z

Z is a high-level specification language: a system can be specified without worrying about details of its implementation. That is, it can be used to specify *what* a system does, rather than *how* it does it. A mathematical specification of a system also allows us to reason about a system, without reference to its implementation. The main feature of the Z language is the schema. Schemas are used to split the specification into parts, which may then be re-used. The schema calculus then allows these parts to be composed.

The next few paragraphs explain how Z is used in the Compliance Notation. Firstly, the specification of an Ada program is written in Z. Typically, a literate script will contain a number of Z paragraphs. When formally specifying a fragment of an Ada program, the user will insert a specification statement in the appropriate place. This specification statement consists of Z and will use definitions from the Z paragraphs.

Z can be used when a specification statement is refined. Refinement introduces programming constructs and possibly more specification statements. These new specification statements might need to refer to some new Z paragraphs. For example, a Z schema can be used to express the invariant of a loop. The name of this schema (a schema reference) can then be used in a specification statement. In the Compliance Notation, Z is also used to express the VCs that are generated during refinement, and to express their formal proof.

Apart from the VCs in a compliance argument, the projected Z document contains a Z translation of the Ada constants, types and functions found in the literate script. These form the Z context of the VCs and are necessary for type checking and proving them.

2.3 SPARK

SPARK [21] is a "safe subset" of Ada. SPARK's restrictions increase the chances of writing a trustworthy program, and at the same time make the formal verification

of programs easier. For example, it rules out those Ada programs of uncertain meaning, which may be interpreted differently by different compilers. Without such restrictions, a programmer could write a program thinking it has a certain meaning, when a compiler gives it another. This restriction also helps with the formal verification of programs easier: showing compliance between a program and its specification is easier if the program has a precise meaning. If there were many possible meanings then each of these possibilities would have to comply with the specification of the program.

An example of an Ada program having more than one meaning is one containing a function with a side-effect. If a function F changes a global variable A, then an expression such as $A + F(B)$ could give different results depending on whether A or $F(B)$ is evaluated first. Two different compilers might therefore produce two different results. Functions with side-effects also cause the order of evaluation of parameters of a subprogram to be important. If two of the parameters are A and $F(B)$, then different results will be obtained depending on which of these parameters is evaluated first. To avoid these problems, SPARK rules out functions with side-effects.

Ambiguity can also arise through aliasing, which may be caused by the use of access types, and so these are banned by SPARK. Aliasing may also be the result of passing actual parameters by reference, and this could give different results to passing parameters by copying in and copying out. SPARK rules out programs that depend on the type of parameter passing.

SPARK rules out programs that use dynamic memory allocation. This avoids programs that run out of memory at run-time; the formal verification of such programs would also be very difficult. SPARK therefore rules out recursion and discriminated record types. Access types also use dynamic memory allocation, and this is another reason why these are ruled out.

SPARK rules out both the re-declaration of identifiers already in scope and overloading. This prevents a programmer becoming a victim of variable capture; without this restriction, a variable different to the one intended could be updated. Programs that allow redeclaration and overloading are more difficult to verify.

2.4 Literate Programming

Literate programming [138] allows the parts of a program to be presented in a different order to that presented to a compiler. A program can be presented in an order more suited to human comprehension, rather than in an order convenient to a machine. For example, it is often easier to understand a program by seeing the overall structure, before concentrating on the individual parts.

To achieve this, literate programming uses a system of slots, known as *Knuth slots*. A slot is used to denote a piece of the program; a piece that will be described later. A slot contains a brief description in natural language of the piece of program that will ultimately occupy the slot. So, an overall description of the program can be achieved before expanding on the individual parts.

A slot also contains a label that is used when the slot is later expanded; it signifies which slot is being expanded. The labels also allow the final program to be extracted in the correct order for the compiler.

For example, a programmable logic controller (PLC) for a metal press can be presented thus:

```
procedure PLC_SYSTEM
is
    ⟨ Declarations ⟩              (1)
begin
    ⟨ Initialisation ⟩            (2)
    ⟨ Non−Terminating Loop ⟩      (3)
end PLC_SYSTEM;
```

This description gives a good overall view of the program, without worrying about how the non-terminating control system is actually implemented; this can be described later.

A Knuth slot, such as (*1*) above, can be expanded to show the piece of program that is to occupy the slot. This will usually be accompanied by a full, natural language description of the code, rather than the brief description that is in the slot. A program fragment that to expand a slot can itself contain further slots, and so on. For example, expanding slot (*1*) gives:

```
(1) ≡
    OPEN, CLOSE : BOOLEAN;
    TIME : INTEGER;
    CLOCK : BOOLEAN;
    type STATETYPE is range 0 .. 7;
    INIT : constant STATETYPE := 0;
    ISOPEN : constant STATETYPE := 1;
    RIGHT : constant STATETYPE := 2;
    LEFT : constant STATETYPE := 3;
    CLOSING : constant STATETYPE := 4;
    U_CLOSING: constant STATETYPE := 5;
    OPENING : constant STATETYPE := 6;
    WAITING : constant STATETYPE := 7;
    STATE : STATETYPE;
    LBD_IN, RBD_IN : BOOLEAN;
    PRESS_OPEN_IN, PRESS_CLOSED_IN,
    PRESS_PONR_IN : BOOLEAN;
```

The declarations for slot (*1*) allow the state for a metal press with two buttons (one for opening the press and one for closing it) to be described. If the variable *OPEN* is set to *TRUE*, then the press will open. If the variable *CLOSE* is set to *TRUE*, then the press will close. *TIME* represents a clock within the controller, while *CLOCK* represents whether it is switched on. *STATETYPE* represents the different states of the press. *LBD_IN* and *RBD_IN* are the left and right buttons; they

are *TRUE* if pressed. *PRESS_OPEN_IN*, *PRESS_CLOSED_IN* and *PRESS_PONR_IN* represent sensor values indicating that the press is fully open, fully closed or past the point of no return respectively.

2.5 Refinement

Refinement [192, 257] is a formal relationship between a program and a formal specification. It allows the formality present in the specification to be continued all the way to code, and allows the task of relating an implementation to a formal specification to be broken down into parts. To achieve its aims, refinement uses specification statements, which are similar to Knuth slots, except this time the slot contains a formal description of code, rather than a brief English description. A specification statement formally describes the code required at that point.

A specification statement can be expanded (refined) into a program, which can contain further specification statements, which can themselves be subsequently refined. Each refinement step may incur VCs, and the program introduced complies with the specification statement provided the VCs hold. VCs can be generated automatically and can be submitted to a theorem prover, just as the final program can be submitted to a compiler.

Refinement is similar to literate programming: it allows the problem to be decomposed, and the program to be presented in a different order to that presented to a compiler; but at the same time it brings formality into the process.

In the Compliance Notation, a specification statement is used to formally specify in Z the Ada code that is required at that point. A specification statement can be placed wherever an Ada statement is allowed, and forms the origin for the code that is ultimately to replace it. As an example, consider:

$$
\begin{array}{l}
\mathit{procedure}\ PLUS\_TEN\ (X : \mathit{in}\ INTEGER;\ Y : \mathit{out}\ INTEGER) \\
\mathit{is} \\
\mathit{begin} \\
\quad \Delta\ Y\ [\ X = 2,\ Y = X + 10\ ] \\
\mathit{end}\ PLUS\_TEN;
\end{array}
$$

The procedure $PLUS\_TEN$ has been specified by its interface (the procedure header) and its effect (a specification statement for its body). It is required to output a number ten greater than its input (a postcondition) whenever the input is equal to two (a precondition); it can of course do the same for other inputs. Only Y is allowed to change (the frame of the specification). One solution to the specification statement is the code $Y := X + 10$, which achieves the postcondition for all inputs. Another solution is $Y := 12$, which achieves the postcondition exactly when the precondition holds; this is the minimum requirement.

An example of a refinement in the Compliance Notation for the previous specification statement is:

$$\sqsubseteq\ \ Y := X + 10;$$

The symbol "$\sqsubseteq$" denotes a formal refinement that incurs a VC that the assignment satisfies the specification statement it replaces.

2.6 A Compliance Tool

The Compliance Tool supports the use of the Compliance Notation. It performs the following principal functions:

- Checking the syntax of a Compliance Notation script.
- Generating the Z document, including VCs, from a script.
- Extracting the SPARK program from a script.

The Compliance Tool is implemented as an application of the ProofPower specification and proof tool. The tool provides all the facilities offered by ProofPower for developing specifications and proofs in HOL and Z, and for preparing high-quality printed output in LaTeX. The tool includes some custom facilities for working with the Compliance Notation, including a VC browser and some customized proof procedures.

The use of the tool in the development of a compliance argument will involve several stages, typically including the following:

- *Initial preparation of literate scripts using the editing and interactive checking facilities of the tool.* During this stage, Z documents and SPARK programs are produced and inspected as required to assist in the development.
- *Batch processing of complete literate scripts.* If the conventions suggested in the User Guide are followed, the Z documents and SPARK program are produced automatically. The tool can be used interactively if required, for example to help diagnose errors.
- *Further analysis of the Z documents.* Depending on circumstances and on the level of formality required, this might involve some or all of the following: inspection of the documents, on paper or using a viewer; use of the VC browser supplied with the tool; and machine checking the proof of some or all of the VCs.

The Compliance Tool offers an extensive range of facilities to help in all three of these activities.

Note that in the example given in 2.5, a side condition to the VC might be a further VC

$$X+10 \in INTEGER$$

which ensures that a run-time error cannot occur. In this example, this would exclude the possibility of exceeding the range of a type. The Compliance Tool deliberately does not generate this side condition as an additional formal VC, because other methods can be used to establish whether it holds.

A static analysis tool called Malporte is used within the Systems Assurance Group to demonstrate absence of run-time errors. The capabilities and use of Malporte are discussed later in the chapter (see Section 6.7), but it supports reasoning about partial program correctness. Therefore, conditions such as loop termination are also not enforced; however, loop termination can be proved separately using the Compliance Tool.

Another example of a compliance argument is the following procedure to swap the values held in two variables:

$procedure\ SWAP\ (X, Y : in\ out\ INTEGER)$
is
$\quad \langle\ Local\ variable\ \rangle \qquad (1)$
$begin$
$\quad \langle\ Swap\ the\ values\ of\ X\ and\ Y\ \rangle \qquad (2)$
$end\ SWAP;$

Expanding slots (*1*) and (*2*), using the expansion symbol $\equiv$, gives:

$(1) \equiv\ \ TEMP : INTEGER;$

and

$(2) \equiv\ \ \Delta\ X, Y, TEMP\ [\ true,\ X = Y_{\mathrm{o}} \wedge Y = X_{\mathrm{o}}\ ] \qquad (3)$

The above specification statement is of the form

$\Delta\ frame\ [\ precondition,\ postcondition\ ]\ (label)$

The postcondition states that the final value of X must equal the initial value of Y (denoted Y_0) and the final value of Y must equal the initial value of X (denoted X_0). The frame states that only X, Y and $TEMP$ are allowed to change. The final step of the compliance argument is a refinement of the previous specification statement:

$(3) \sqsubseteq$
$\quad TEMP := X;$
$\quad X := Y;$
$\quad Y := TEMP;$

The refinement of the program has now reached code.

3 Experiences of Using the Notation and Tool

Experience gained from using the Compliance Notation and the Compliance Tool has led to four observations about how they should be used.

The first observation is that writing an abstract specification, although very valuable, is difficult. People seem to be able to write programs that, more or less, meet a requirement much more easily than they can write an abstract specification. For this reason, it is easier to get people to write concrete programs and then abstract them. If this is done the program that "more or less" meets a requirement changes to a correct program, with a better appreciation of how design decisions can affect a requirement.

The second observation is that concrete Ada variables and types should be used unless an abstract type is genuinely easier to understand. Novices to Z do tend to write specifications that look like programs, hence guidance on writing

Z specifications tends to encourage the use of abstract mathematical types, such as sets and relations. Unfortunately, more experienced Z practitioners can fall into the trap of introducing false abstraction.

For example, if a specification is written in terms of a set and operations on sets, rather than a sequence and sequence operations, then a judgement has to be made about whether the use of a set is very much clearer than a sequence.

Using a set is a little more abstract than using a sequence, but if the specification is to be refined to an array, then there will be a heavier penalty in terms of effort and cost in performing data refinement if we start from a set. If an abstraction gives a much clearer specification, then this penalty can be worth paying; however, experience has shown that for control system applications this is the exception rather than the rule.

The third observation is the converse of the second, specifiers can fall into the trap of leaving a small gap between the specification and the implementation. In this case, the benefit gained from reading the specification rather than the source code is outweighed by the great deal of effort and money required to demonstrate formal refinement.

It has been observed that industrial users tend to describe the total functionality of a software component rather than concentrate on critical properties. This can be cost effective, but for certification it is often the case that only certain critical safety properties need to be demonstrated. These safety properties can be identified using a range of hazard analysis techniques on the total system.

Traditionally, a specification was proved to satisfy identified safety properties, and then code was refined from the specification; however, as the third observation shows, this can lead to little extra benefit for a lot of cost.

The fourth observation is that, when applicable, safety properties should be formally demonstrated directly from the code. It is still worth checking the specification rigorously to demonstrate that the safety properties are satisfied, in order to reduce the economic risk of producing uncertifiable software. It is not however cost effective to attempt relatively expensive formal proofs at the level of the specification when the certifier wants assurance that the code is safe.

3.1 An Example

The programmable logic controller, described in Section 2.4, is used to illustrate these observations, before we relate more experiences of the actual industrial usage of the Compliance Notation and Tool. Recall that the declaration slot in the overall program skeleton was expanded to a sequence of Ada declaration statements. These Ada declarations are automatically translated into Z, enabling the safety condition to be stated.

This specification gives necessary and sufficient conditions for when the metal press should be open and closed. As far as the safety of the operation of the press is concerned, this is all that needs to be demonstrated. Note that the Ada variables *STATE*, *OPEN* and *CLOSE* appear as Z identifiers in the schema *PLC*, whereas the Ada constants *INIT*, *OPENING*, *CLOSING* and *U_CLOSING*

appear as Z constants (these constants will be automatically declared in the extracted Z document, so that the schema type checks).

z

PLC

$$STATE : STATETYPE;$$
$$OPEN, CLOSE : BOOLEAN$$

$$OPEN = TRUE \Leftrightarrow STATE \in \{INIT, OPENING\};$$
$$CLOSE = TRUE \Leftrightarrow STATE \in \{CLOSING, U\_CLOSING\}$$

The initialisations slot (*2*) in the program skeleton must establish the safety property, and this is formalized.

$$(2) \sqsubseteq \ \Delta\, CLOSE, OPEN, STATE, TIME\ [\ PLC\ ]\ \ (4)$$

This states that only *CLOSE*, *OPEN*, *STATE* and *TIME* can be changed in subsequent refinements, and that the postcondition denoted by *PLC* must be established with the precondition of *true*. The refinement of this generates VCs ensuring that the initialisations establish the safety condition.

$$(4) \sqsubseteq$$
$$CLOSE := FALSE;$$
$$OPEN := TRUE;$$
$$TIME := 0;$$
$$STATE := INIT;$$

The loop slot (*3*) in the program skeleton can be expanded, but it must always preserve the safety condition.

$$(3) \equiv$$
loop
$$\Delta\, CLOSE, OPEN, STATE, TIME\ [\ PLC, PLC\ ]\ \ (5)$$
end loop;

The verification conditions that are generated for this step ensure that the safety condition is maintained.

The issue of which actual button pushes and sensor inputs produce which actions, can just as easily be read from the code as from some Z. The Ada comments for each arm of the if-statement below help in this task. Even if Z were used to specify this, it would not add extra assurance, because the Z would exactly mirror the Ada. The task would then be to check the Z, since the VCs produced could be automatically proved. Hence the literate script is not complicated with Z at this point. Also a benefit of this is to reduce the number of VCs. If each arm of the if-statement were a specification statement, then extra

VCs would be produced; therefore, code is introduced instead.

$$
\begin{array}{l}
(5) \sqsubseteq \mathit{if} \quad -- \mathit{reset\ PLC} \\
\quad \mathit{LBD\_IN} = \mathit{FALSE}\ \mathit{and}\ \mathit{RBD\_IN} = \mathit{FALSE}\ \mathit{and} \\
\quad \mathit{PRESS\_OPEN\_IN} = \mathit{TRUE}\ \mathit{and} \\
\quad (\mathit{STATE} = \mathit{INIT}\ \mathit{or}\ \mathit{STATE} = \mathit{WAITING}) \\
\mathit{then} \\
\quad \mathit{STATE} := \mathit{ISOPEN}; \\
\quad \mathit{OPEN} := \mathit{FALSE}; \\
\quad \mathit{CLOSE} := \mathit{FALSE}; \\
\mathit{elsif}\ -- \mathit{left\ button\ pressed} \\
\quad \mathit{LBD\_IN} = \mathit{TRUE}\ \mathit{and}\ \mathit{STATE} = \mathit{ISOPEN} \\
\mathit{then} \\
\quad \mathit{STATE} := \mathit{LEFT}; \\
\quad \mathit{TIME} := 500; \\
\mathit{elsif}\ -- \mathit{right\ button\ pressed} \\
\quad \mathit{RBD\_IN} = \mathit{TRUE}\ \mathit{and}\ \mathit{STATE} = \mathit{ISOPEN} \\
\mathit{then} \\
\quad \mathit{STATE} := \mathit{RIGHT}; \\
\quad \mathit{TIME} := 500; \\
\mathit{elsif}\ -- \mathit{left\ button\ pressed\ (right\ button\ already\ pressed)} \\
\quad \mathit{LBD\_IN} = \mathit{TRUE}\ \mathit{and}\ \mathit{STATE} = \mathit{RIGHT} \\
\mathit{then} \\
\quad \mathit{STATE} := \mathit{CLOSING}; \\
\quad \mathit{CLOSE} := \mathit{TRUE}; \\
\mathit{elsif}\ -- \mathit{right\ button\ pressed\ (left\ button\ already\ pressed)} \\
\quad \mathit{RBD\_IN} = \mathit{TRUE}\ \mathit{and}\ \mathit{STATE} = \mathit{LEFT} \\
\mathit{then} \\
\quad \mathit{STATE} := \mathit{CLOSING}; \\
\quad \mathit{CLOSE} := \mathit{TRUE}; \\
\mathit{elsif}\ -- \mathit{left\ button\ released} \\
\quad \mathit{LBD\_IN} = \mathit{FALSE}\ \mathit{and}\ \mathit{STATE} = \mathit{LEFT} \\
\mathit{then} \\
\quad \mathit{STATE} := \mathit{ISOPEN}; \\
\mathit{elsif}\ -- \mathit{right\ button\ released} \\
\quad \mathit{RBD\_IN} = \mathit{FALSE}\ \mathit{and}\ \mathit{STATE} = \mathit{RIGHT} \\
\mathit{then} \\
\quad \mathit{STATE} := \mathit{ISOPEN}; \\
\mathit{elsif}\ -- \mathit{left\ button\ timed\ out} \\
\quad \mathit{TIME} = 0\ \mathit{and}\ \mathit{STATE} = \mathit{LEFT} \\
\mathit{then} \\
\quad \mathit{STATE} := \mathit{WAITING}; \\
\mathit{elsif}\ -- \mathit{right\ button\ timed\ out} \\
\quad \mathit{TIME} = 0\ \mathit{and}\ \mathit{STATE} = \mathit{RIGHT} \\
\mathit{then} \\
\quad \mathit{STATE} := \mathit{WAITING};
\end{array}
$$

$$
\begin{array}{l}
\quad elsif \;\; -- \; press \; is \; beyond \; the \; point \; of \; no \; return \\
\quad\quad PRESS\_PONR\_IN = TRUE \; and \\
\quad\quad STATE = CLOSING \\
\quad then \\
\quad\quad STATE := U\_CLOSING; \\
\quad elsif \;\; -- \; press \; open \\
\quad\quad PRESS\_OPEN\_IN = TRUE \; and \\
\quad\quad STATE = OPENING \\
\quad then \\
\quad\quad STATE := WAITING; \\
\quad\quad OPEN := FALSE; \\
\quad elsif \;\; -- \; press \; closed \\
\quad\quad PRESS\_CLOSED\_IN = TRUE \; and \\
\quad\quad STATE = U\_CLOSING \\
\quad then \\
\quad\quad STATE := OPENING; \\
\quad\quad OPEN := TRUE; \\
\quad\quad CLOSE := FALSE; \\
\quad elsif \;\; -- \; abort \; close \\
\quad\quad PRESS\_PONR\_IN = FALSE \; and \\
\quad\quad (LBD\_IN = FALSE \; or \; RBD\_IN = FALSE) \; and \\
\quad\quad STATE = CLOSING \\
\quad then \\
\quad\quad STATE := OPENING; \\
\quad\quad OPEN := TRUE; \\
\quad\quad CLOSE := FALSE; \\
\quad end \; if; \\
\quad \Delta \; TIME \; [\; PLC, \; PLC \;] \qquad (6)
\end{array}
$$

This single refinement step is very convenient, but it creates fifteen VCs; fortunately, they are all easily discharged by ProofPower.

Setting the clock (6) is done in a separate refinement step to reduce the number of VCs; otherwise, the number of VCs for step (5) would be the product of the number of paths through the two if-statements below. This way, the number of VCs is the sum of the number of paths through the two if-statements.

$$
\begin{array}{l}
(6) \equiv if \; CLOCK = TRUE \\
\quad then \\
\quad\quad if \; TIME = 0 \\
\quad\quad then \\
\quad\quad\quad TIME := TIME; \\
\quad\quad elsif \; TIME > 0 \\
\quad\quad then \\
\quad\quad\quad TIME := TIME - 1; \\
\quad\quad end \; if; \\
\quad end \; if;
\end{array}
$$

Although a fragment of a compliance argument has been presented in a top-down manner, it is actually easier to write an operational program. Think of the operation of the metal press:

> *If the left button is pressed, then record this in a state variable and set the countdown timer to 500; if the right button is pressed, then do the analogous action; if the left button is pressed and the right button has already been pressed then...*

It seems more natural to think operationally, and more difficult to create a functional specification that is significantly more abstract. Of course, this is not universally true, but it seems to be a general rule for control systems. It is easier to try to link the program with an abstract critical safety property identified from a hazard analysis.

The compliance argument used translations of the Ada variables into Z; these identifiers were used in the specification of the safety property without any loss of clarity. Thus, false abstraction was avoided: with the consequent penalty of data refinement, the code was shown to directly satisfy the identified critical property; and the gap between the specification and code was large enough to make the effort worthwhile and not completely obvious in the first place.

4 Industrial Usage of the Compliance Tool

4.1 SHOLIS

The acronym SHOLIS stands for *Ship Helicopter Operating Limits Instrumentation System*; it is a computer aide to the safe landing and take-off of helicopters from ships. One of the most important functions of SHOLIS is to raise an alarm when a helicopter is operating outside its safe envelope for take-off or landing. It is also important that false alarms do not occur, because this would impair the safe recovery of an airborne helicopter. Of course, SHOLIS would also become ineffective if people lost their trust in it.

Twice a second SHOLIS does the following: it obtains environmental information from external sensors; it checks whether the current conditions are outside the pre-defined safety envelope; and it outputs the results by raising or removing alarms. Safety properties are that, for each type of alarm, the output of SHOLIS corresponds correctly to sensor inputs.

4.2 The Compliance Argument

The application software of SHOLIS amounts to approximately 25,500 non-blank, non-comment lines of SPARK. A compliance argument for about 20% of the software was produced under an evaluation contract from the Systems Assurance Group. The argument demonstrated that, for one of the three alarm types, the output of SHOLIS did correspond correctly to the sensor inputs. This 20% was a coherent subset that contributed to the safety property and was

distributed through the software; that is, it was not a segregated kernel. The compliance argument amounted to nearly 750 pages.

The UK software engineering company Praxis produced a software requirements specification in Z as part of their normal software engineering practice. The Z specification was used as a basis for all the software development, and all the application software can be traced to elements of the software design, which in turn can be traced to the software requirements specification. The software requirements specification was also used as part of the system hazard analysis to derive formal expressions of safety properties; that is, a partial, formal specification that captures that property the software must satisfy if it is not to contribute to a system-level hazard. No changes were made to the SHOLIS software to facilitate the production of the compliance argument, since the evaluation was not part of the main development project. The argument was fitted retrospectively onto the code produced by the SHOLIS development. Praxis did not have any prior knowledge of the Compliance Notation however they do have extensive experience of Z, SPARK and the SHOLIS application. In addition some informal training and support was given to Praxis by SAG.

Unsurprisingly, the size of the compliance argument was determined by the number of lines of code included and the degree of formality in the presentation. For the largest package considered, 1,350 lines of non-blank, non-comment code, the argument was approximately 200 pages, and the majority of the compliance argument was formal refinement. The Z specification of the package accounts for about 40 of the 200 pages; the refinement from specification statements to code was almost 100 pages.

4.3 Experiences

By allowing Z specifications to be included as part of a compliance argument, the Compliance Notation provides considerable flexibility in the style of presentation of individual specification statements. Schemas are especially useful for encapsulating the detail in specification statements and maintaining clarity.

It was found easy within the SHOLIS evaluation to vary the level of formality. The only threads of control considered within the SHOLIS software were those that achieved the two safety properties. As a consequence, formal compliance arguments were needed only for those subprograms that contributed to the safety properties; in other areas, the presentation was informal. In fact, Praxis were able to produce a formal argument that focused on the specific threads of control through certain subprograms, without needing to extend the formal treatment to the remaining threads of control. This clearly demonstrated that the Compliance Notation is capable of providing required and varied levels of formality.

One of the design objectives for the Compliance Notation and Tool was to be independent of the development process. The evaluation was performed independently of the SPARK programs independently from how they were developed.

Praxis have found Z to be a very expressive and flexible formal notation. Praxis found during the SHOLIS evaluation that, within a package, the Compliance Notation generally made good use of Z to provide an effective way of both specifying subprograms and performing formal refinement.

Furthermore, SPARK provided by the Compliance Notation intuitive and easy to use. There were, however, a number of issues concerning the visibility of Z and Ada definitions that limited the kinds of Ada subprograms that could directly be used as part of a compliance argument. Fortunately, work-arounds were found during the evaluation and the Compliance Notation and Tool have been modified to eliminate these problems in the present tool. The problems arose essentially out of scaling a compliance argument up from one or two packages to a dozen or more. Without a tool and a real example to apply it to, it is very difficult to anticipate all the visibility issues surrounding Z and Ada.

Proving VCs provides indisputable evidence that the Ada source code is compliant with its Z specification. During the SHOLIS evaluation, it was found that ProofPower, upon which the compliance tool is built, was very effective at discharging the generated VCs. No conflict was found between the style of Z used in specification statements and the ease of proof of the subsequent verification conditions with ProofPower.

ProofPower was also very effective at schema manipulation, and allowed exactly the required amount of detail to be uncovered by selective schema expansion. Generated VCs can be extremely large, but they are subsequently reduced substantially by a single simple rewrite. For example the update of a large deeply nested data structure gave rise to a VC of 8.5 Mbytes, but this was reduced to a page and a half by a single rewrite application.

Figure 1 includes approximate sizes for the documentation and code produced by the SHOLIS software development, and estimates for the corresponding aspects of a complete compliance argument. The estimates are based on the size of the partial compliance argument produced for the SHOLIS evaluation.

	SHOLIS development	**Compliance Argument**
Specification	350pp.	—
High-level design	50pp.	50pp.
Detailed design	550pp.	100–200pp.
Code	25.5kloc	3,500–4,000pp.

Fig. 1. Documentation size for SHOLIS

The SHOLIS project's high-level design presents the software architecture (the Ada package structure) and covers general software issues, such as event scheduling and diagnostics. The detailed design documentation describes the state and subprograms of each Ada package, and includes information such as precise memory maps and justification of numerical accuracy. The detailed designs do not contain any source code listings, formal subprogram specifications, or any derivation or refinement of the subprograms.

4.4 Results

A compliance argument to replace the SHOLIS software design documentation, which contained all the SHOLIS code, would need to include the same high-level design information; however, much of the detailed design is inherently part of a compliance argument by its very nature of documenting the code. The detailed design figure given for the compliance argument in Table 1 represents information, such as precise memory maps and justification of numerical accuracy. The code figure given for the compliance argument is based on the 750 pages contained in the partial compliance argument.

The partial compliance argument produced during the evaluation is highly formal, the majority of subprograms have specification statements that are formally refined to code; however, only a few example proofs were carried out, which corresponded to about 10% of the total number of pages. The sheer bulk of the compliance documentation does raise a slight concern about how practical it is to use. Praxis accepted that this is inevitable when considering the complete behaviour of the given amount of software and the Compliance Notation provided an effective way of structuring the volume of information.

Caution is needed when comparing the effort spent on producing the partial SHOLIS compliance argument with the actual SHOLIS development. This is because the compliance argument was largely produced when the SHOLIS design documentation and code was already written. In addition, the Z specification of the SHOLIS software provided a very useful starting point for writing specification statements for subprograms. The effort spent producing the compliance argument was mainly casting the code into a literate script and producing concrete Z specifications and refinement steps; nearly 20% of the effort was spent producing example proofs.

Given this background the effort required to produce a complete compliance argument for SHOLIS was estimated by a simple *pro rata* extrapolation from the effort for the partial compliance argument, excluding proof. The result is that a complete argument would take a little less effort than it took to design and write all the software, with limited testing. The resolution of the issues that required "work-arounds" would decrease the effort to produce an argument.

5 Control Law Verification

Tools such as Simulink are being used by engineers around the world to model and solve real problems concerning dynamical systems. Once a control system model has been developed, an implementation of the control system can be automatically generated. This facility is very useful: both industry and government regulators are striving for a reliable method for writing code to reduce lengthy validation tests. In December 1998, the French civil certification agency DGAC approved the autopilot for the Eurocopter EC 155 helicopter, which had its software automatically generated by the CS Verilog "Scade" development environment. The EC 155 autopilot was Level A, the most critical to flight safety.

Level A requires extensive testing that, even when no errors are found, is expensive. Testing also does not provide the same degree of evidence of correctness of the automatically generated code as proof does. The very high reliability claims required for avionics systems and the role of testing have given rise to controversy in the past and are discussed in [41, 157].

There is a significant gap between the code generated and the control law diagram in Simulink. The diagram is understood by a control law engineer; but once it is transformed into code, it is no longer readily understandable.

There are three possible approaches to solving this problem: follow a method of "correct-by-construction"; demonstrate that the implementation of the autocoder is correct; or demonstrate compliance between a control law representation and code generated from it, for each critical application. The correct-by-construction approach was investigated under a contract from the UK MoD [209]. A functional specification representing the control law diagram was transformed into code using already proven laws. This is an ambitious approach that is equivalent to developing a refinement calculus for control law diagrams. It was unclear to what degree the transformations could have been automated, but there was potential for a very useful tool for cost effective verified code generation. Unfortunately, the commercial market is such that tools such as Simulink will be more widely adopted. This is because of the extra and improved facilities that they will provide on a regular basis in response to the market.

If the demonstration that an autocoder is correct is to be carried out to the same level of rigour as the correct-by-construction approach, then mathematical proof is required. This is an arduous task of the same level of difficulty as the correct-by-construction approach. This approach also suffers from the competition from COTS tools such as Simulink.

The final approach of demonstrating compliance between an individual control law and its implementation is at least an order of magnitude easier, but needs to be repeated for each control law. This involves constructing a proof based on the structure of the code and the control law diagram. The advantage of tool-generated code is that it is generated in a consistent manner, meaning that machine support can be developed that will ease the burden of proof, in this limited domain, quite considerably. The main advantage of this approach is that it can be used with a COTS tool.

5.1 Control Law Diagrams

The class of control laws considered in this section is that of PID (Proportional Integral Derivative) controllers. A PID controller is a general purpose controller that can be tuned to the particular system under control. A PID controller can be used even when the differential equations governing the dynamics of the system are unknown. In such circumstances the controller can be tuned by running the actual system and observing the output.

Figure 2 shows how a PID controller fits into the control of a system. A PID controller controls the value of a controlled variable by adjusting the value of a manipulated variable. The main input to a PID controller is the current error,

which is the difference between the demanded value of the controlled variable and its actual value. The output of the PID controller is the value of the manipulated variable required to reduce the error.

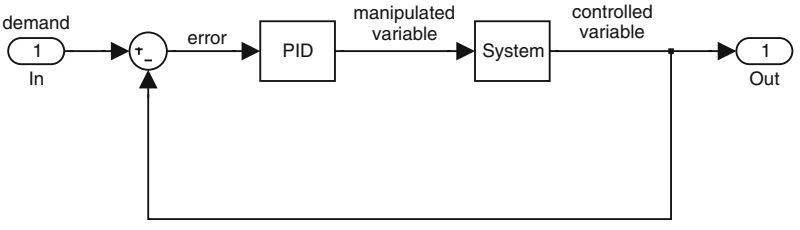


Fig. 2. General PID controller

A PID controller consists of three parts: the proportional, integral and derivative contributions to the value of the manipulated variable. The proportional contribution, as its name suggests, is proportional to the error input. The integral contribution is designed to remove long-term errors by forming the sum of the errors so far. This sum continues to rise until the long-term error has disappeared. The derivative contribution is a measure of how fast the error is changing, and so makes the controller more responsive.

The PID controller considered here controls a Fuel Metering Valve (FMV). The main input is the position error of the valve (the difference between the demanded and measured position), and the output is the electric current to be input to the valve's motor to reduce the error. The example is taken from [209], but with the simplification that the control law runs in a single lane.

Figure 3 shows the results of simulating the PID controller using Simulink. Two output graphs are shown: the current input to the valve's motor and the position of the valve. The demanded position of the valve is 45°, as shown. The remaining inputs to the PID controller (the constant boxes shown) are arbitrary for the purposes of simulation; the actual values would depend on the particular aircraft that the engine is powering. The output graphs show that the position of the valve does settle to the demanded value.

5.2 The Diagram

The figure in Appendix C shows the control law diagram for the PID controller, drawn using Simulink. There are eight inputs to the control law and one output. The inputs are lozenge shaped boxes numbered one to eight, and the output is the lozenge shaped box on the far right of the diagram. The main input is FMVPE, the position error of the valve. The output FMTMCD is the electric current to be input to the valve's motor to reduce the error.

There are ten constants on the diagram: the square-shaped boxes containing numbers. The triangular boxes denote a multiplying operation, the *gain*, by

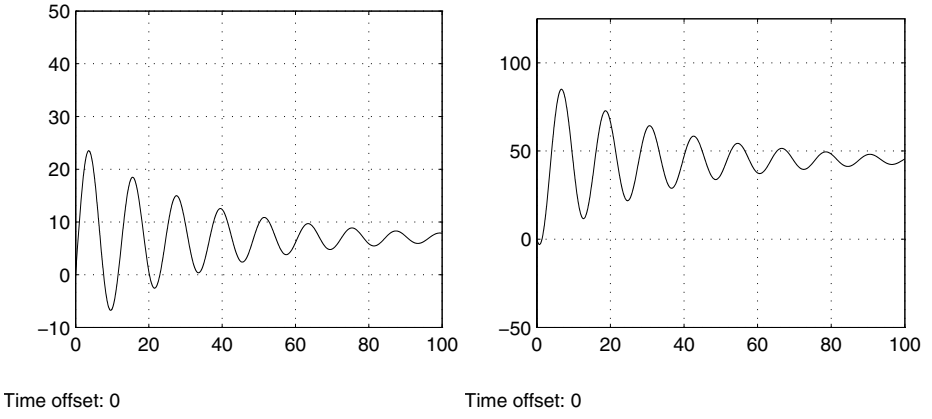


Fig. 3. Simulation results for the FMV

the number inside the box. The rectangular/square boxes containing names are subsystem boxes. They have not been further expanded, but they could be, using the same approach for each subsystem. The boxes containing a 1/z symbol are unit-delay boxes: they store their input on one cycle and output it on the next.

The reason there are Add and Sub subsystem boxes, instead of standard addition and subtraction points, is that these subsystems, if expanded, would limit their results in order to avoid exceptional behaviour.

5.3 A Method for Compliance

The problem to be addressed is how to formally link a control law diagram with the code generated from a tool. The answer lies in the observation that code is generated for each control block, and then brought together according to the signal paths, the *wiring*, in the diagram. To provide an argument for compliance that is independent of the particular version of the tool, the structure of the diagram and generated code must be used as the starting point.

1. Provide the overall structure of the compliance argument, based on the code structure, using Knuth slots.
2. Expand any Knuth slots for constant declarations, in order to automatically declare Z constants that will be available for use within the specification.
3. Define a Z schema for each control block.
4. Define a schema *Diag* that is the conjunction of all the schemas describing blocks within the diagram.
5. Define a schema *Internals_sig* that contains all the intermediate signals.
6. Define the procedure that implements the control law diagram with the specification statement:

$$\Delta\, STATES,\ OUTPUTS\ [\ \exists\ Internals\_sig\ \bullet\ Diag\ ]$$

7. If necessary, perform refinement steps to complete compliance argument.

8. If a control block is itself a subsystem, then this method can be followed recursively, with a renaming scheme to avoid variable capture if other subsystem control blocks are present.
9. The schema *Internals_sig* is the collection of names and types representing all the signals within the diagram. The conjunction of all the individual schemas representing blocks is the mechanism that wires the blocks together in the way represented by the diagram.

5.4 A Case Study

In this section, the method for generating a Z specification from a control law diagram, which is then refined to code, is illustrated using the PID controller. The code used here was not generated by the Real Time Ada Workshop tool within Matlab, for reasons of clarity, conciseness and some minor syntactic issues. There are more significant issues discussed in the conclusions, but the code that was generated is not fundamentally different enough to invalidate this case study.

The code consists of an Ada package: a specification and a body. The package specification contains a procedure *PID*, which is the Ada procedure that is called for each cycle through the control law. For the purposes of this chapter, a procedure to initialize the control law has not been considered.

Below is an overview of the Ada package body. It contains ten Knuth slots, giving the overall structure, and containing a brief description of the Ada that ultimately occupies that slot.

```
package body PID_PACKAGE
is
    ⟨ control law constants ⟩                (1)
    ⟨ state for unit delay boxes ⟩           (2)
    ⟨ differentiator subsystem box ⟩         (3)
    ⟨ integrator subsystem box ⟩             (4)
    ⟨ scale subsystem boxes ⟩                (5)
    ⟨ limit subsystem boxes ⟩                (6)
    ⟨ add subsystem boxes ⟩                  (7)
    ⟨ sub subsystem box ⟩                    (8)
    ⟨ upper subsystem box ⟩                  (9)
    ⟨ body of procedure pid ⟩                (10)
end PID_PACKAGE;
```

This provides a link between the control law diagram and the code that can be reviewed by a human. Although this is a creative step, it is considered relatively easy, and is the step 1 in the compliance method.

Slots (1) to (9) ultimately contain Ada that is called by the body of procedure PID in slot (10). For example slots (3) to (9) contain the Ada subprograms implementing the subsystem boxes of the control law diagram; these subprograms are called by procedure *PID*.

The following Ada constants implement the constant boxes on the control law diagram; these are the square boxes containing numbers. This is step 2 in the compliance method.

```
(1) ≡
    DFMVGD : constant integer := 1;
    SCFDIF : constant integer := 25;
    DFGAIN : constant integer := 25;
    DFMVGP : constant integer := 1;
    SCFPRP : constant integer := 500;
    SCFINT : constant integer := 1250;
    ITGAIN : constant integer := 2048;
    SCFPRF : constant integer := 500;
    SMPIDO : constant integer := 2;
    SDPIDO : constant integer := 1;
```

Z constants are generated by the tool when this is processed, and they are globally available within the Z document. The following sections describe the specification of some key blocks within the diagram, as an example of step 3 in the compliance method.

5.5 Scale Boxes

The Ada function *SCALE* implements the Scale boxes. It is an Ada stub because, as discussed earlier, the subsystem boxes on the control law diagram have been left unexpanded.

z

$$scale : \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$$

```
(5) ≡
    function SCALE (
            INPUT : integer;
            MULT : integer;
            DIVISOR : integer) return integer

        Ξ [ SCALE (INPUT, MULT, DIVISOR)
                        = scale (INPUT, MULT, DIVISOR) ]
    is separate;
```

The value returned by this Ada function is defined in terms of a Z function *scale*, whose semantics are given, for completeness, in Appendix C.

In the Compliance Notation, a formal annotation is used to specify the semantics of Ada functions. In this example, the annotation is the following specification statement:

$$\Xi\ [\ SCALE\ (INPUT,\ MULT,\ DIVISOR) \\ = scale\ (INPUT,\ MULT,\ DIVISOR)\ ]$$

Any non-local variables that the Ada function reads would be listed after the Ξ symbol. In this case there are no such variables; if the Scale subsystem box was expanded, then it would not use any state. The predicate in brackets constrains the value returned by the Ada function, stated in terms of *scale*. The code in the corresponding sub-unit that implements the scale subsystem must satisfy this constraint. Verifying the compliance between subsystem specifications and their implementations usually incurs VCs. It could be the case, for some control laws, that the subsystems are trusted components and that verification of the subsystems would, by the ALARP principle, not be cost effective.

When the compliance tool processes the above Ada stub a Z function *SCALE* is generated. This Z function is an abstraction of the Ada stub and can be used to write the following Z schema.

z

Scale1

$$FMVPE,\ SCDIF : \mathbb{Z}$$

$$SCALE\ (FMVPE,\ DFMVGD,\ SCFDIF) = SCDIF$$

This is an abstraction of part of the code, targeted at the Scale1 box of the control law diagram; in this way, parts of the Ada can be linked with parts of the control law diagram. Thus, the Z schema boxes map to the pictorial representation of the control law block diagram. Note that the identifier *DFMVGD* that appears in the schema *Scale1* is not declared within the schema. This is because it is a constant whose Z definition was automatically generated by the Ada declaration of constants earlier in Knuth slot (*1*). The other Scale boxes are similar.

5.6 Unit Delay Boxes

The following Ada variables implement the state required for the unit-delay boxes. *STATE1* is the Ada state required for the UnitDelay1 control-law box and *STATE2* is the Ada state required for the control law box named UnitDelay2.

```
(2) ≡
      STATE1 : integer;
      STATE2 : long_integer;
```

The implementation assumes that an *integer* is a single length word (16 bits) and *long_integer* is a double length word (32 bits). The double length word is used as an input to the integrator, whereas the single length word is used as an input to the differentiator (see later).

The box UnitDelay1 stores its input SCDIF. In the Compliance Notation, $STATE1_0$ denotes the initial value of the state at the start of the control law cycle, and *STATE1* denotes the value at the end of the cycle. The predicate in the Z schema below states that, at the end of the control law cycle, the input to a unit delay box, *SCDIF*, is held in the state.

z

UnitDelay1

$$STATE1,\ STATE1_{\mathrm{o}} : \mathbb{Z};$$
$$SCDIF : \mathbb{Z}$$

$$STATE1 = SCDIF$$

Similarly the box UnitDelay2 holds its input *LINRES*.

z

UnitDelay2

$$STATE2,\ STATE2_{\mathrm{o}} : \mathbb{Z};$$
$$LINRES : \mathbb{Z}$$

$$STATE2 = LINRES$$

5.7 Differentiator Box

The Ada function *DIFFERENTIATOR* implements the differentiator subsystem box; again it is an Ada stub.

z

$$differentiator : \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$$

```
(3) ≡
    function DIFFERENTIATOR (
            INPUT : integer;
            LAST_INPUT : integer;
            GAIN : integer) return integer
        Ξ [ DIFFERENTIATOR (INPUT, LAST_INPUT, GAIN)
                = differentiator (INPUT, LAST_INPUT, GAIN) ]
    is separate;
```

The differentiator computes the rate at which the error is changing (the derivative) by comparing the error input on this cycle with the previous error.

The error input to the differentiator on this cycle is *SCDIF* and the previous error is held in the state variable $STATE1_0$ of the box UnitDelay1.

z

Differentiator

$SCDIF, FMVDTM : \mathbb{Z}$

$DIFFERENTIATOR\ (SCDIF, STATE1_o, DFGAIN) = FMVDTM$

This is an example of an implementation feature moving up the compliance route toward the specification and simplifying the code-to-specification mapping.

5.8 Integrator Box

The Ada function *INTEGRATOR* implements the integrator subsystem box; it computes the sum of the errors (the integral) from this and the previous cycles by adding the error input on this cycle with the previous integral. The integral is always held as a *long_integer*.

z

$integrator : \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$

(4) ≡

function INTEGRATOR (
PRV_INTEGRAL : *long_integer*;
INPUT : *integer*;
GAIN : *integer*) *return long_integer*

Ξ [$INTEGRATOR\ (PRV\_INTEGRAL, INPUT, GAIN) = integrator\ (PRV\_INTEGRAL, INPUT, GAIN)$]

is separate;

z

Integrator

$INTINPUT, INTRES : \mathbb{Z}$;
$STATE2_o : \mathbb{Z}$

$INTEGRATOR\ (STATE2_o, INTINPUT, ITGAIN) = INTRES$

The error input to the integrator on this cycle is *INTINPUT* and the previous integral is held in the state variable $STATE2_0$ of the box UnitDelay2.

5.9 Complete Control Law Diagram

This subsection describes steps 4 to 7 in the compliance method. An abstraction of the code of the Ada procedure PID is provided, which is targeted at the complete control law diagram; this abstraction is formal and incurs a VC. Following this specification statement, the code that implements it is given.

First the schemas describing the individual blocks are conjoined together using schema inclusion.

z

Diag

$$Scale1;\ Scale2;\ Scale3;\ Scale4;$$
$$UnitDelay1;\ UnitDelay2;$$
$$Differentiator;\ Integrator;$$
$$Add1;\ Add2;\ Add3;\ Add4;$$
$$Limit1;\ Limit2;\ Upper;\ Sub$$

Note that this technique of schema inclusion, which conjoins the schemas together, works because Z identifies duplicate identifiers, which is in effect joining together the named wires in the diagram. This completes step 4 in the compliance method. Step 5 can be done easily by a machine, and the result is the following schema containing all the internal signals—the names for the internal wires in the diagram.

z

Internals_sig

$$SCDIF,\ FMVPTM,\ INTINPUT : \mathbb{Z};$$
$$FMVFTM,\ SINPUT,\ LIMINP : \mathbb{Z};$$
$$LINRES,\ FMVDTM,\ INTRES : \mathbb{Z};$$
$$FM1MX,\ FM1MN,\ FMVER1 : \mathbb{Z};$$
$$FMVER2,\ ULINRES : \mathbb{Z}$$

The body of procedure *PID* can now be given, according to step 6 in the compliance method. A specification statement is embedded in the body, just before the keyword '*is*'. This specification is an abstraction of the code of procedure *PID*, targeted at the complete control law diagram. The specification statement states that the code satisfies schema *Diag*, and changes only the state variables *STATE1* and *STATE2*, and the output *FMTMCD*. All the internal intermediate signals within the diagram are hidden by existential quantification, leaving only input and output signals visible, as well as the state used to record unit delays.

```
(10) ≡
     procedure PID (
          FMVPE : in integer;
          DFMVGI : in integer;
          FMVPV : in integer;
          DFMVGF : in integer;
          DFM2MN : in integer;
          DFM2MX : in integer;
          CFMCMX : in integer;
          CFMCMN : in integer;
          FMTMCD : out integer)

        Δ STATE1, STATE2, FMTMCD [∃ Internals_sig• Diag ]

     is
        SCDIF, FMVDTM, FMVPTM, INTINPUT,
        FMVFTM, FM1MN, FM1MX, ULINRES,
        FMVER1, FMVER2, SINPUT, LIMINP : integer;
        INTRES, LINRES : long_integer;

     begin
        SCDIF := SCALE (FMVPE, DFMVGD, SCFDIF);
        FMVDTM := DIFFERENTIATOR (SCDIF,
                              STATE1, DFGAIN);
        STATE1 := SCDIF;
        FMVPTM := SCALE (FMVPE, DFMVGP, SCFPRP);
        INTINPUT := SCALE (FMVPE, DFMVGI, SCFINT);
        FMVFTM := SCALE (FMVPV, DFMVGF, SCFPRF);
        FM1MN := ADD (DFM2MN, FMVFTM);
        FM1MX := ADD (FMVFTM, DFM2MX);
        INTRES := INTEGRATOR (STATE2,
                          INTINPUT, ITGAIN);
        LINRES := LIMIT1 (INTRES,
                    65536 * long_integer (FM1MX),
                    65536 * long_integer (FM1MN));

        STATE2 := LINRES;
        ULINRES := UPPER (LINRES);
        FMVER1 := SUB (ULINRES, FMVFTM);
        FMVER2 := ADD (FMVER1, FMVPTM);
        SINPUT := ADD (FMVDTM, FMVER2);
        LIMINP := SCALE (SINPUT, SMPIDO, SDPIDO);
        FMTMCD := LIMIT2 (LIMINP, CFMCMX, CFMCMN);

     end PID;
```

This completes step 6 in the compliance method, incurring a single VC that was discharged using ProofPower. For more complicated control laws and code, the implicit refinement step would have to be split into a number of smaller steps, which constitutes step 7. The initial refinement step would contain a mixture of Ada code and further specification statements, which would then be further refined. This would result in a number of simpler VCs being generated.

6 A System Case

DO178B is a development process advocated by agencies such as the FAA, and can be used to produce software that is acceptable for non-life-critical applications. Level A of DO178B is intended for use in safety-related applications, but it can be argued that in the civil domain DO178B is not used for safety-critical systems. In this chapter, software is considered safety critical if its incorrect operation would lead to the loss of life, and there is no other mitigation possible (such as the intervention of a pilot).

Safety-critical software is becoming more prevalent in the defence domain, and it is only a matter of time before it enters the civil domain. Safety-critical software requires a high degree of assurance before it can be accepted by a responsible procurer. This has led to the definition of standards like Defence Standard 00–55 in the UK. There are other similar standards across the world, and they all take the same basic approach: they advocate a particular form of software development using formal methods.

When an organisation can adopt this very stringent form of development, it seems to lead to a highly assured product. The problem is that the development processes in these standards are so stringent that they seem to lead to "gold plated" developments that are disproportionately more expensive and risky from a project management point of view.

Different organisations have broadly similar development processes, but the details of those processes can be quite different. These differences are due to differences in training, domain of the applications, regulatory requirements and culture, to name just a few. The more controlled and rigorous the development required for higher assurance, the more difficult it seems to be to adopt widely.

6.1 The Steam Boiler

The steam-boiler problem has been fully described in [4]. It can be briefly outlined as a control issue, with water being removed from a boiler as steam, at a varying but measured rate, and replaced by the action of four identical pumps. A controller aims to hold the water level within allocated bounds *N1* and *N2* (see Figure 4), and to shut the system down if there is a risk that the water level could go outside the wider bounds *M1* and *M2*. There may be failures in the individual pumps, the water level measurement device and the steam measurement device. The steam-boiler system has been simulated using the Matlab program

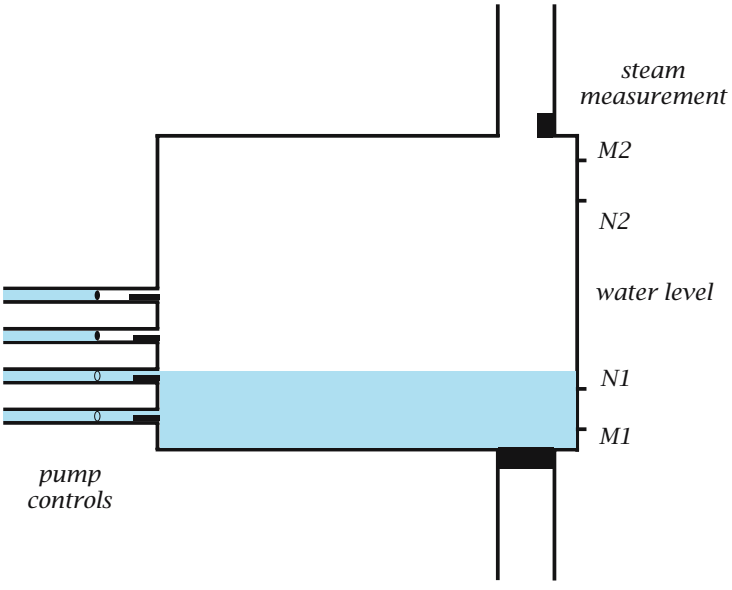


Fig. 4. The steam boiler

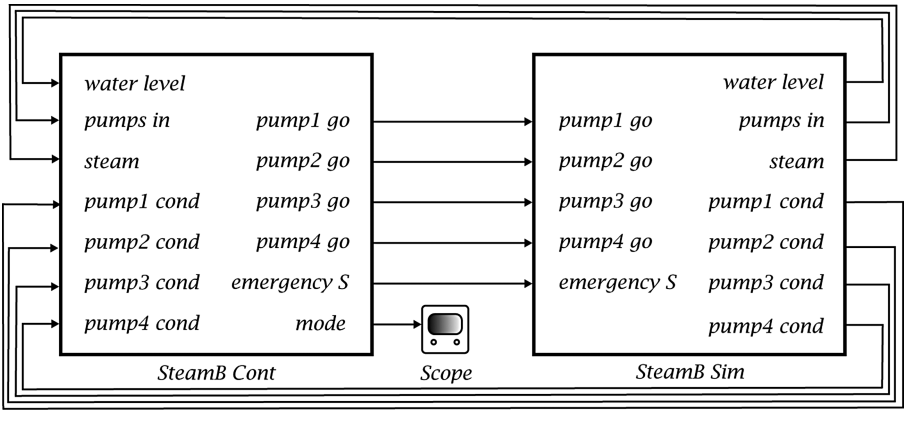


Fig. 5. The steam-boiler simulation

Simulink. The simulation models the steam boiler and its controller as two separate entities. These are shown in Figure 5. The steam-boiler entity consists of a simulated steam outflow and an inflow provided by the pumps. The outflow rate has a maximum value of F units per second and a minimum of zero. The increment is a randomly chosen value between G and $-G$. Each pump has a pumping capacity of P units per second when working, and zero when not working. The level measurement device determines the water level in the boiler; it may also be determined by reference to a previous value, together with a subsequent flow

history. Failure and repair of pumps and measuring devices have been simulated using random numbers.

The steam-boiler controller entity was simulated using four functional blocks. These are the level-decision block, the pump-activation block, the mode-identifier block and the emergency-stop block. The steam-boiler controller is shown in Figure 6. All outputs from the controller were subject to a delay of one second to provide synchronous operation of the system.

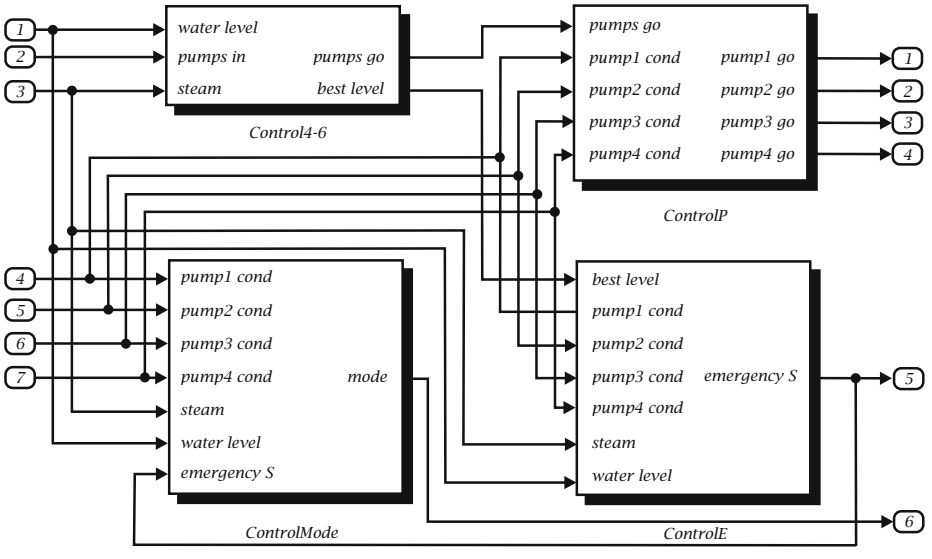


Fig. 6. The steam-boiler controller

6.2 An Acceptance Argument

Goal Structuring Notation (GSN) [251] provides a diagrammatic view of the safety case for a system, in this case for the steam boiler. One use is to supply references to individual safety arguments, assumptions and constraints, which can then be examined in detail for weaknesses. A second use is to make it possible to specify a case for acceptance at successive levels, and to check for omissions and shortcomings. For instance, the safety case in Figure 7 could be amplified to distinguish between the verification of Ada code and of the native code targeted at a selected microprocessor. This approach could be taken for any acceptance property, such as security or reliability.

Figure 7 shows the top-level goal G0001 and its deconstruction into subgoals. Each dark ellipse indicates a subgoal to be addressed by safety arguments developed outside of the methods described in this chapter. Subgoal G003 is the one at second level that is refined in Figure 7. Subgoals G0008 and G0009 are developed further in Figures 8 and 9. Figure 7 shows the development of subgoal

G0008 for the software specification. Subgoal G0016 is the only subgoal in the figure addressed by the processes of this chapter.

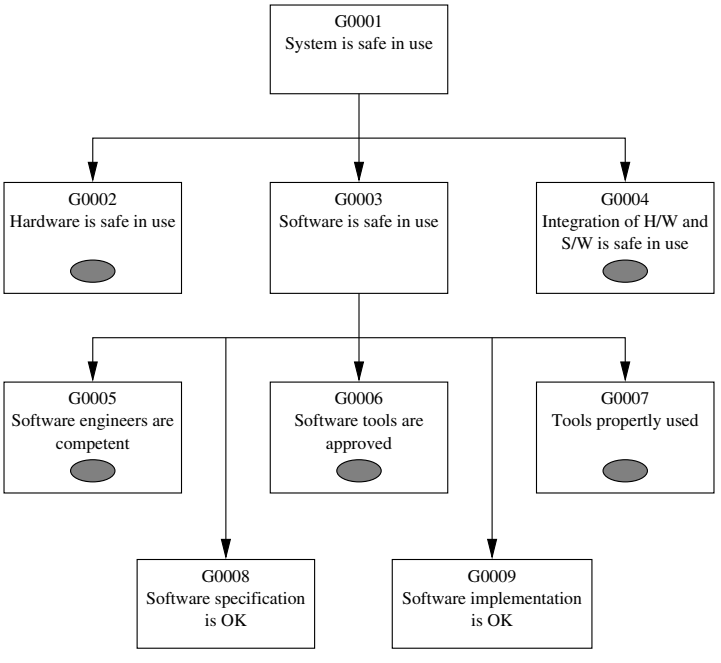


Fig. 7. Breakdown of top-level goal G0001 into subgoals

6.3 Goal 0016: Identifying Safety Properties

It is relatively straightforward to identify system hazards; the difficulty comes in propagating the system hazards down to the level where software and hardware operate. In practice, using fault trees to link system hazards to software becomes intractable below a physical component, such as a Digital Engine Control Unit (DECU). For example, a fault-tree analysis would identify that the DECU should not shut down the engine, unless instructed by the pilot. If it were not explicitly part of the overall DECU design, it could be quite difficult to justify that the software would not cause an undemanded shutdown using fault trees.

The use of Failure Modes Effects Analysis could start with how possible software faults could manifest themselves as system failures. Unfortunately, this quickly becomes intractable for any realistic software system.

The identification of safety properties for software is crucial to limiting the scope of evidence required to show that a system is safe enough to accept. If safety properties cannot be identified, then the developer and customer are trapped into demonstrating full functional correctness. Further, they still require an analysis that the functionality is sufficient for safety. Unfortunately, it has to be recognized that this is impossible where the safety property is an emergent property

of a complex control function. For example, another safety property of a DECU is that it should not cause an engine to run so fast that it becomes damaged. To separate this out from the rest of the control function would be intractable for a complex control law. In practice, the software would have to be shown to accurately implement the control law, and to rely on control theory evidence and testing to accept that the DECU was safe.

Nancy Leveson's work [153] on requirements specification for black-box behaviour provides one means of identifying whether software components are critical. Another complementary approach is under development within the Systems Assurance Group at Malvern. The basic idea is to assess a system in terms of the services provided by its components, treated like Leveson as black boxes; they can therefore be either COTS or bespoke. It is assumed that the control function is subject to the following non-exclusive failure modes:

- Omission failure of a service.
- Commission failure of a service.
- Early or late delivery of a service.
- Data corruption of a service output.

This approach means that lower-level failures can be abstracted away. For example, a run-time error in software could cause: the loss of a service (omission); the service to do something it should not do (commission); corruption of output data; and even late or early delivery of a service.

In this approach, the exact cause of a service failure is not of interest; what is of interest is its consequence with respect to system safety. Failure Modes and Effects Analysis is similar in spirit, but crucially the abstraction to failures of service means that it can be made tractable. It is still too complicated to be done manually, but automated methods using model checking have been developed.

For the steam-boiler case study, the following contextual information is relevant:

- The controller shuts system down if the water level is within 5 seconds of boiling dry or overflowing; this is the "water-limit risk" level.
- The failure of pumps can be non-manifest (not detected by the controller).
- The water-level sensor is polled every 5 seconds.
- It takes 5 seconds to pressurize the pumps to add more water.

Model checking against the system safety property that the water must always stay within the minimum and maximum limits produced a violation. The following accident scenario was returned:

- The water level fell to a point where the pumps had to be activated.
- The water level then rose and eventually the pumps were shut down.
- At this point the pumps failed silently (a non-manifest failure).
- The sensor polling of the water level was such that the level was just above the "water-limit risk" level.
- Before the level was next sensed the boiler blew up.

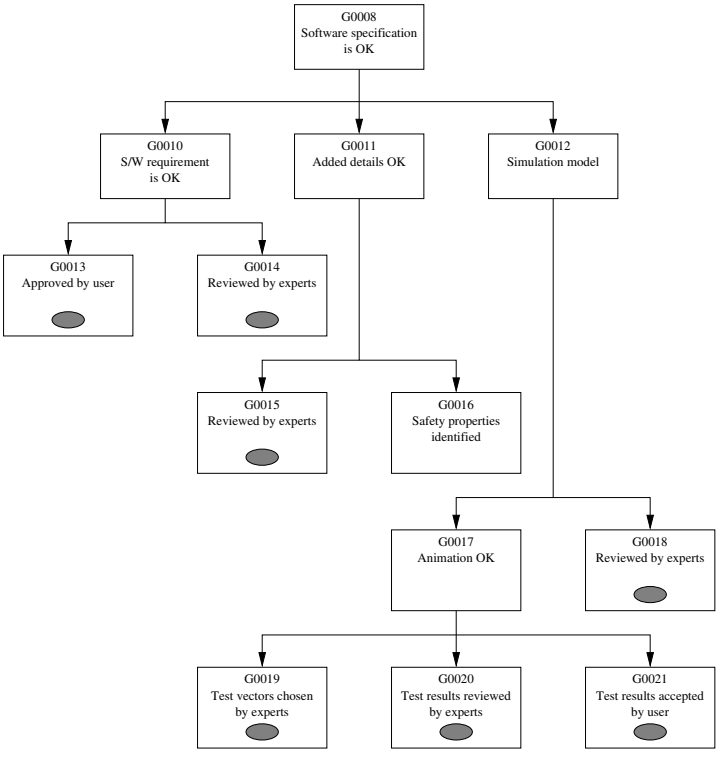


Fig. 8. Breakdown of top-level goal G0008 into subgoals

This is a classic example of a phasing problem in control systems. New safety limits for "water-level risk" were defined to remove this safety risk and the check repeated. The analysis showed that there was only one single point of system failure for omission type failures. This was when the software moding function that instigates an emergency shutdown is faulty then system failure results.

A deeper analysis revealed that there was only one pair of omission failures that could produce a system failure. If the software functions for determining the risk of a safety breach are both incorrect then system failure results. Of course in practice these software functions would probably be the same, therefore this was again a critical common mode failure. This is an important reminder that further analysis is needed to make sure that different black box functions are not vulnerable to a common mode failure. Interpreting the analysis also indicated that the reliability of the sensors could largely determine whether a numeric safety target for a system could be met.

The verification of safety properties for control systems has been described earlier in this chapter. The next section tackles a more difficult problem, where the safety property cannot be teased out from the overall functionality of the control function, as in the case of over-speed of an engine controlled by a DECU.

6.4 Goal G0009: Software Is OK

Figure 9 shows the development of subgoal G0009 from Figure 7 for the software implementation. The figure shows how the acceptance based verification techniques, described later, contribute to the integrated verification activity to create assurance. The task of accepting the software as being safe breaks down

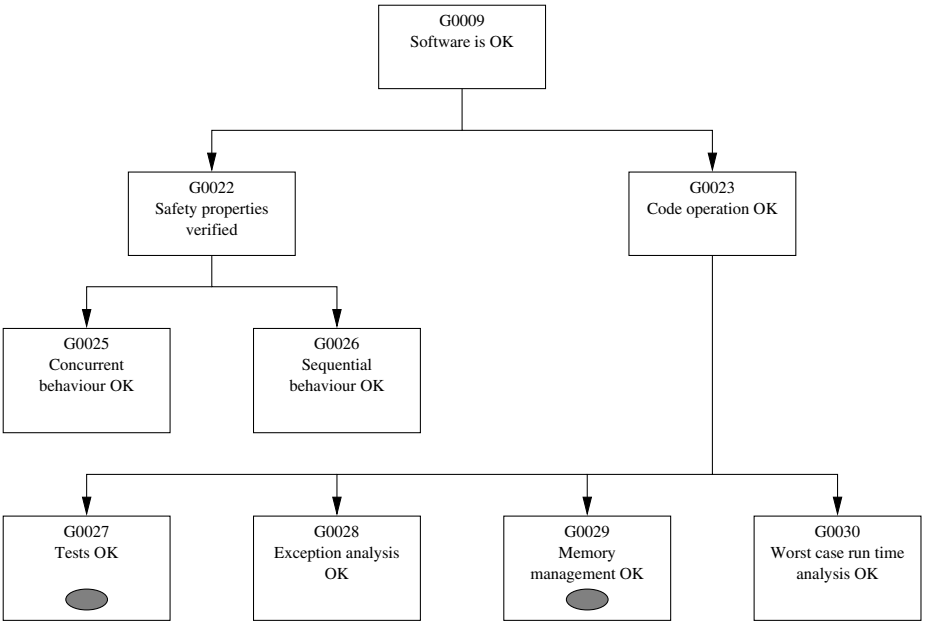


Fig. 9. Further breakdown of goal G0009 into subgoals

into six acceptance tasks. Four of these are described later; the remaining two, testing and memory management, are not covered in this chapter. The acceptance projects that the Systems Assurance Group has been involved in, have relied on compiler evidence for the acceptance evidence for memory management. The compilers had to be for suitable languages (such as Ada) and be rigorously tested and established.

6.5 Goal G0026: Sequential Behaviour OK

Simulink has been used earlier as a means of validating the development of a control system through simulation. The diagrams produced using Simulink can also provide a specification for a software controller.

ClawZ [11] (pronounced 'claws') is a tool that links Simulink diagrams with Z[257]. In particular this provides a bridge between the use of Simulink to define control law diagrams and the use of the DAZ tool [206, 207] that verifies

refinement conjectures. ClawZ operates by translating a Simulink model into a Z specification that captures the functionality of the Simulink model. This Z specification can then be used in conjunction with a library of supporting definitions to construct a refinement conjecture, or compliance argument, which can then be formally verified using ProofPower. A detailed description of how the Z is produced is described in [11], and the form of a refinement conjecture (between the Z representation of a control law diagram and its implementation in the SPARK subset [21] of Ada) was given earlier in this chapter.

Subsequent work has developed a process to automatically generate a refinement conjecture between the Z representation and the Ada implementation, using a tool called RSG *(Refinement Script Generator)* tool. It links subsystems of a diagram with a fragment of sequential Ada code, called the witness, which is that part of the software that implements that subsystem. This is a manual process that requires human intelligence; however, it is relatively simple, and non-specialists in formal methods have been trained within a week or two to carry out this task.

The refinement conjecture is processed by the DAZ tool, which in turn automatically generates VCs that verify that the Ada code does indeed implement the subsystem. The VCs, as before, are discharged using ProofPower.

The ClawZ, RSG, DAZ and ProofPower tools have all been used on the Simulink description of the steam-boiler controller to verify automatically generated code. These tools have not been used as part of the development: only after the code has been generated have they been employed as part of a software acceptance process.

This acceptance technique has also been used on manually developed code from another organisation. It again proved to be very effective. The errors that were found were easily corrected.

The success of this acceptance technique for sequential code relies crucially on the software being systematically developed. A software engineering process of transforming a specification into code is essential. If such a development process is not followed, then the likely outcome is that the verification will fail so badly that it would be cheaper to re-develop the software. Without a systematic development process then the verification is reduced to the 1970's "guess and verify" approach. Indeed the RSG tool essentially reflects one particular way software can be systematically developed. Other variants of the RSG tool would be required for significantly different, but equally valid, systematic software development processes.

RSG—Abstracting Analysis for Simulink Specifications

The analysis of sequential code of a control system is achieved by showing that the code is correct for a single execution cycle. The analysis starts with the following components:

- The control law, for example the Simulink diagram shown in Figure 10.
- A program that implements the control law, as shown in Figure 11.

– A link between control law inputs/outputs and program inputs/outputs, referred to as an *interface*, as shown in Figure 12. Note that *System* is a schema that is the translation of the Simulink diagram (in Figure 10) into Z by ClawZ. The ClawZ output is shown in Figure 13.

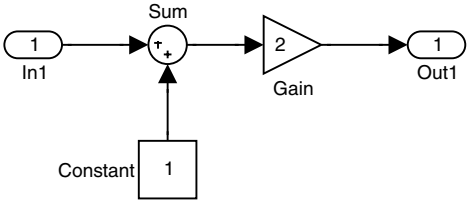


Fig. 10. 'System'—a Simulink model

```
procedure Step (X : in Float; Y : out Float)
is
   TMP : Float;
begin
   TMP := X + 1.0;
   Y := 2.0 * TMP;
end Step;
```

Fig. 11. An implementation of the Simulink model 'System'

In a typical development of a control system, all three components will be present. In defining the requirements, a control law is expressed in Simulink and validated by simulation, providing the Simulink diagram. As part of the program development, the software detailed design will map out the relationship between signals in the control law and program variables, providing the interface. The development will result in a program that implements the control law.

A Z specification for the procedure is systematically derived from the interface by defining the postcondition

z

$$Post \mathrel{\widehat=} \exists\, System \bullet Interface$$

By hiding *System*, *Post* (as a predicate when written in a specification statement) is a predicate on Ada variables that relates outputs to inputs. Allowing only the output variables to change ensures that any implementation of this specification computes the outputs. Therefore *Frame* is defined as follows:

z

$$Frame \mathrel{\widehat=} Outputs$$

z

Inputs

$X : FLOAT$

z

Outputs

$Y : FLOAT$

z

Interface

System;
Inputs;
Outputs

$In1? = X$;
$Out1! = Y$

Fig. 12. Simulink—Ada Interface

Annotating the procedure *Step* (see Figure 11) with this formal specification statement gives

```
procedure Step (X : in Float; Y : out Float)
    Δ Frame [ Post ]
is
    TMP : Float;
begin
    TMP := X + 1.0;
    Y := 2.0 * TMP;
end Step;
```

At this stage DAZ could be used to generate VCs that, if proved, would show compliance; however, there is a far more intuitive, though less formal, justification that the Ada program implements Figure 10.

From the interface in Figure 12, we see that *In1* corresponds to X in the program; thus, by the wiring of the Simulink diagram, input 1 of *Sum* also corresponds to X in the program. Similarly, output 1 of *Constant* corresponds to *1.0* in the program, and so does input 2 of *Sum*.

Since the block *Sum* in Simulink and the '+' operator in Ada perform the same function, we can argue that the output of *Sum* corresponds to *TMP* in

z

$$System\_Constant \mathrel{\widehat=} Constant\ (Value \mathrel{\widehat=} 1\ e\ 0)$$

z

$$System\_Gain \mathrel{\widehat=} Gain\_I\ (Gain \mathrel{\widehat=} 2\ e\ 0)$$

z

$$System\_Sum \mathrel{\widehat=} Sum\_P2$$

z

System

$In1? : \mathbb{U};$
$Constant : System\_Constant;$
$Gain : System\_Gain;$
$Sum : System\_Sum;$
$Out1! : \mathbb{U}$

$Out1! = Gain.Out1!;$
$Sum.In2? = Constant.Out1!;$
$Sum.In1? = In1?;$
$Gain.In1? = Sum.Out1!$

Fig. 13. ClawZ translation of 'System'

the program after the first statement. Then, by the wiring of the diagram again, input 1 of *Gain* also corresponds to *TMP* before the second statement.

So we can argue that the output of *Gain* corresponds to Y in the program after the second statement. As the output of *Gain* is connected to *Out1* then *Out1* corresponds to Y at the end of the program.

For those Simulink signals mentioned in the interface (*In1*? and *Out1*!), the correspondences between Simulink signals and program variables satisfy the interface. Therefore we can argue that the program is compliant.

In summary, for every input and output of every Simulink block, we have found a corresponding expression at some point in the program.

Looking back at the specification statement, consider the proof requiring that the program implies the postcondition. (Note that this is not the verification condition that is generated.)

$$TMP = X +_R 1\ e\ 0,\ \ Y = 2\ e\ 0 *_R TMP\ \ ?\vdash\ \ \exists\ System \bullet Interface$$

These expressions in the Ada program that correspond to Simulink signals are precisely the existential witnesses to *System* in the consequent; hence they are

referred to as witnesses. The justification above can be expressed more succinctly by annotating a program with witnesses as follows. Here the symbol '⟵' means 'is witnessed by'.

$$
\begin{array}{l}
\quad \{\ In1? \longleftarrow X, \\
\quad\ \ Sum.In1? \longleftarrow X, \\
\quad\ \ Constant.Out1! \longleftarrow 1.0 \\
\quad\ \ Sum.In2? \longleftarrow 1.0\ \} \\
\\
TMP := X + 1.0; \\
\\
\quad \{\ Sum.Out_1! \longleftarrow TMP, \\
\quad\ \ Gain.In_1? \longleftarrow TMP\ \} \\
\\
Y := 2.0 * TMP; \\
\\
\quad \{\ Gain.Out_1! \longleftarrow Y, \\
\quad\ \ Out_1! \longleftarrow Y\ \}
\end{array}
$$

Clearly, this notation cannot be checked by DAZ, as it is informal; however, these annotations can be formalized according to a systematic process that abstracts statements to specification statements, and captures these witnesses in their pre and postconditions.

To accommodate this systematisation, witnesses are given either before or after a statement; therefore, witnesses after the first statement are distinct from those before the second statement. Secondly, witnesses are actually Z expressions, where Ada variables are referred to in exactly the same way as for pre and postconditions. We now have

$$
\begin{array}{l}
\quad \{\ In1? \longleftarrow X, \\
\quad\ \ Sum.In1? \longleftarrow X, \\
\quad\ \ Constant.Out1! \longleftarrow 1\ e\ 0 \\
\quad\ \ Sum.In2? \longleftarrow 1\ e\ 0\ \} \\
\\
TMP := X + 1.0; \\
\\
\quad \{\ Sum.Out1! \longleftarrow TMP\ \} \\
\\
\quad \{\ Gain.In1? \longleftarrow TMP\ \} \\
\\
Y := 2.0 * TMP; \\
\\
\quad \{\ Gain.Out1! \longleftarrow Y, \\
\quad\ \ Out1! \longleftarrow Y\ \}
\end{array}
$$

A detailed description of the full process is beyond the scope of this chapter. In brief, each statement is abstracted to a specification statement, where witnesses

given before it are captured in the precondition, and those given afterwards are captured in the postcondition. Witnesses are accumulated working from the top. Once all witnesses for a block's inputs and outputs have been given, a *block predicate* is introduced. In the above example, witnesses have been given for all inputs (there are none) and outputs of *Constant* before the first statement. Therefore, the precondition contains the following block predicate:

$$\exists\, Constant : System\_Constant \bullet Constant.Out1! = 1\ e\ 0$$

A block predicate is simply a predicate on Ada variables (possibly none) that uses (the Z translation of) a Simulink block to encapsulate some functionality. (Note that *Post* is itself a block predicate, just defined using schemas.)

After the first statement, witnesses have been given to all inputs/outputs of *Sum*. The entire statement is abstracted to the following specification statement:

$$\begin{aligned}
&\Delta\, TMP\ [\\
&\quad \exists\, Constant : System_Constant \bullet Constant.Out1! = 1\ e\ 0,\\
&\\
&\quad (\exists\, Constant : System_Constant \bullet Constant.Out1! = 1\ e\ 0)\\
&\quad \land\ (\exists\, Sum : System_Sum \bullet\\
&\qquad Sum.In1? = A \land\\
&\qquad Sum.In2? = B \land\\
&\qquad Sum.Out1! = TMP)\ ]
\end{aligned}$$

If any property of the first statement was needed later (by a block whose witnesses where not all given yet) then the analyst should explicitly add this property to the postcondition, even if this property could be deduced from the block predicate.

The second statement is abstracted to

$$\begin{aligned}
&\Delta\, Y\ [\\
&\quad (\exists\, Constant : System_Constant \bullet Constant.Out1! = 1\ e\ 0)\\
&\quad \land\ (\exists\, Sum : System_Sum\\
&\qquad \bullet\ Sum.In1? = A\\
&\qquad \land\ Sum.In2? = B\\
&\qquad \land\ Sum.Out1! = TMP),\\
&\\
&\quad \exists\, System\\
&\qquad \bullet\ In1? = X\\
&\qquad \land\ Gain.In1? = TMP\\
&\qquad \land\ Gain.Out1! = Y\\
&\qquad \land\ Out1! = Y\ ]
\end{aligned}$$

The block predicate for *System* uses schema quantification, as there is no block name for the entire system. *Sum* and *Constant* are not mentioned, as witnesses need to be stated only once, and *System* encompasses their functionality. Note

that the postcondition is very similar in structure to *Post*, and does indeed imply *Post*.

There are many other aspects to this process not mentioned here. It is worth mentioning just one though: when stepping over a statement that updates a program variable X, any witness in the accumulated pool that refers to X can no longer do so. Therefore a logical constant is introduced in the refinement and substituted for occurrences of X in witnesses.

The resulting refinement, though less readable to the untrained eye, has one very important advantage over a hand-written refinement: the VCs produced are much more amenable to automatic proof, because there is a greater number of smaller conjectures. This is due to the detailed matching of functionality between the Simulink and Ada program that is easily expressed using witnesses.

This process is used by the RSG to generate refinement scripts (though its interface for giving witnesses is more powerful than annotating the program with every witness). The VCs produced by DAZ from refinement scripts created by RSG are submitted to a very powerful, but highly bespoke, proof tool called *Supertac*. This has two main parts: the first is tailored to the RSG and removes the block predicate structure, leaving shorter logical and arithmetic conjectures about ClawZ and Compliance Notation functions, which are then tackled by the second part.

Using the RSG in conjunction with Supertac has several important advantages over manual refinement and proof:

- De-skills analysis: reasoning is performed at a higher level without the need to know about refinement, although a basic knowledge of Z is still required. Consequently, the bulk of the analysis can be performed by less-skilled people.
- Design evolution: localized changes to the control law and the program result in localized changes to the witnesses. The same is not always true of a refinement, because there is often repetition, for example, as predicates are carried through pre and postconditions of specification statements. Therefore, a localized change to the control law and the program can require changes to the refinement in many places and adjustments to many proofs.
- The notation is more succinct and quicker to produce—consider the effort require to write proofs manually using a theorem prover.
- Simpler notation has less room for mistakes.
- Witnesses provide traceability from Ada back to the Simulink diagram.

6.6 Goal G0025: Concurrent Behaviour OK

The Simulink description is inherently concurrent, and so it could be implemented concurrently to increase response time to retain control of a system operating in the real world. For example, in Figure 6 the execution of the block "Control4-6" can occur on one processor at the same time as the execution of the block "ControlMode". The two blocks do not communicate between each other, although they do share common inputs. The execution of the block "ControlE"

cannot correctly occur until both the blocks, "Control4-6" and "ControlMode", have completed because "ControlE" requires outputs from both of these blocks.

The verification of the implementation of control law diagrams can easily be split into independent verifications of sequential and concurrent behaviour. The acceptance verification for the sequential behaviour of the steam boiler has been described previously. The acceptance verification technique for a possible concurrent implementation is the subject of this section.

To perform the verification, a specification is first required. A prototype tool to automatically generate the CSP specification, based on ClawZ, has been developed. The tool translates a Simulink diagram into a description that forms part of a CSP specification.

The specification is obtained by translating the Simulink representation into a set of functions relating inputs to outputs. These functions are fed into a CSP model that simulates all possible concurrent executions of subsystems within the Simulink diagram. In terms of the diagram the model will only allow subsystems to execute when either that subsystem's inputs are inputs to the whole diagram or are outputs from another subsystem that has already executed within the model.

Many hard real-time systems employ a restricted form of concurrent execution. For the steam-boiler case study, a particular model has been adopted and an automated verification technique developed. The particular model of execution is synchronous cyclic execution distributed over a number of processors that communicate through shared memory.

The shared memory implements the wires between one concurrently executing subsystem of the diagram and another. In CSP, the model of the implementation is the parallel composition of processes representing a cyclic executive. These processes synchronize on the *tock* event, capturing their synchronous execution.

The FDR refinement model checker [225] is used to verify that the distributed scheduling is a possible behaviour of the specification. In principle, the automated verification could be extended for more sophisticated scheduling policies and multiple clocks. This acceptance technique was trivially employed for the steam-boiler problem, but it has also been used on a very large project.

6.7 Goal G0028: Exception Analysis

The verification, conducted under goal G0026, assumed that the Ada was free from exceptional behaviour. This is usually addressed by programming practices and programming languages; however, this does not guarantee that some exceptional behaviour has not slipped into the code. A developer can annotate the code with numerous simple verification assertions, but this is currently limited to the SPARK subset of Ada and limits the number of developers that can perform this task.

Alternatively Abstract Interpretation can be used to check that the software delivered is free from exceptional behaviour [75, 252]. The Malporte tool has been developed by the Systems Assurance Group to be a tool that can be used during acceptance, or development.

The aim of the Malporte tool is to identify by static analysis those states of a program in which the rules of the coding language may be violated. For a given test on a given program state, the analyser will come up with one of the following conclusions:

- No exceptional or undefined behaviour is possible.
- Exceptional or undefined behaviour is possible.
- Malporte is unable to determine whether exceptional or undefined behaviour is possible.

The tool has been designed to test statically for the following classes of run time errors:

- Overflow and underflow.
- Divide by zero.
- Indexing beyond array bounds.
- Use of unset variables.
- Dereferencing of pointers that contain no data.
- Accessing a variable that has gone out of scope.
- Conversion of types on assignment.

After a run of the Malporte tool, the code locations should be examined where the analyser has identified constraints on the acceptable values of variables. In some instances, it will be found through use of reasoning, unavailable to the analyser, that assignment of values that violate the constraints is not possible. The analysis thus leads to three categories of constraint:

- Positive constraints, which indicate that if the program goes outside the constrained values, an error will occur.
- False constraints, which arise due to approximations in the algorithm, so that an error can never occur.
- Unknown constraints, where due to the complexity of the code it is uncertain into which of the above two categories the constraint falls.

Attention is drawn to the locations of potential errors. Whilst these kinds of errors can sometimes be picked out using run time checks, the aim is to predict such errors statically, and hence show that such errors are not possible for any execution conditions of the program.

The Malporte tool could have been applied to the steam-boiler system; however, the system is too simple to be of interest. For instance, there are no divisions, so divide-by-zero is not an issue. Also there are no arrays, pointers or type conversions. The system would have needed to be made artificially more complicated to be useful.

6.8 Goal G0030: Worst-Case Execution Time

Worst-Case Execution Time (WCET) analysis computes upper bounds on the execution time of tasks in a system. In any system where computing resources

may prove inadequate, concern arises over the run times of individual tasks; this concern is all the greater when safety is involved. Thus, a means of gaining an accurate understanding of run times is essential.

Although WCET analysis can be used as an acceptance technique, it is crucial that it is performed during development. There is no chance that hard real-time systems can be developed and successfully delivered without some form of WCET; however, WCET analysis is also essential for acceptance. The following describes a generic approach for WCET analysis that supports the acceptance process.

WCET analysis may be done at low level, taking account of hardware features such as caching and pipelining. The work described here is at a high level and focuses on high-level language features such as loops and subprogram calls. It has been performed for the Systems Assurance Group by the Computer Science Department of the University of York.

The specific feature of WCET analysis under investigation is symbolic execution. The aim is to get a tighter estimation by characterising the context in which the code is executed. The scheme relies on the fact that not all calls of subroutines and loops take the same time to execute. Instead of the traditional approach by which a worst case is assessed for each subroutine or loop call and used every time, an expression for the run time is constructed that depends on the parameters at run time.

Traditionally, WCET tools have been confined to a single source language, a specific compiler, a particular version of the compiler and of its switches, and a unique board layout and configuration. A lack of portability has hindered take-up of established WCET techniques. Under one of the Systems Assurance Group's research projects there has been work at the University of York aiming to improve on this situation by devising a timing analysis scheme based on Java Byte Code (JBC) [25]. Besides Java, many other languages, including Ada and C, can be compiled to JBC. The JBC may be interpreted or compiled down to native assembler/machine code.

The Java Virtual Machine (or the instructions of JBC) is stack based and has simple address modes. All operations are performed through the operand stack. This allows an efficient implementation of the virtual machine on processors that have few registers and few addressing modes. There are three main types of implementations of the virtual machine: interpreted, just-in-time compiled and ahead-of-time compiled. Of these, only the ahead-of-time complied machine shows the performance and predictability suitable for real-time systems.

In order to achieve predictability the source code program must be predictable (no unbounded loops), any library code must have bounded worst-case execution times and the compiler itself must be predictable. The WCET is computed by considering how long each JBC instruction takes to execute. This in turn depends on how the virtual machine is implemented. A timing model is required of the virtual machine for each particular target the code may run on.

The ideas expressed above are being implemented at York in a prototype tool called Javelin [26]. It has not proved practical to use the Javelin tool on the

steam-boiler example. The tool is being developed initially for C, with Ada to follow, whereas the steam-boiler example uses Ada. The steam-boiler example has a cycle time of 1 second based on the time constants of the system. It is therefore not the kind of time-critical real-time system that would benefit from the use of the tool. Further, the steam-boiler system has not been allocated a processor or board configuration; however, the WCET tool is being designed so as to be compatible with application to the steam-boiler system.

7 Experience of Acceptance Based Assurance

7.1 Background

Some of the techniques described in this chapter have been used on actual procurement projects to accept systems on behalf of the UK MoD. They have all been deployed on a current major procurement in some form or another. The particular procurement cannot be identified, but the current experience of accepting a safety critical system is reported.

The system consists of approximately 180,000 lines of non-comment, non-blank Ada. It is a hard real-time system distributed over six processors executing concurrently in synchrony.

7.2 Identifying Safety Properties

The overall system hazard analysis is being used to determine lower level software safety properties. For example, if a built-in test detects a failure, then it must propagated to the controller so that a reversionary mode is activated. CSP has been used to model the reversionary handling in a way related to that described earlier. Its correctness depends upon the correct functioning of some software functions. Hence software safety properties for these functions have been derived.

7.3 Sequential Code Verification

Most of the current verification effort has been directed towards the implementation of a complex non-linear control law. There is some verification of specific safety properties, such as those for reversionary moding.

The specification of the control laws is estimated to be of the order of 800 pages of A4. It is hierarchical and has been transcribed from a representation in FORTRAN. The transcription process includes a significant validation effort that takes advantage of Simulink's simulation capabilities. The specification in Simulink is suitable for review by independent control engineers, which the previous representation was not, although it could be executed.

The implementation of the control law consists of approximately 18,000 lines of non-blank, non-comment code, distributed over three processors interacting concurrently. Although the control law implementation is only about 10% of the total code, it is the most complex and least well understood. The rest of the code

on the remaining processors consists of: actuation software, that is deterministic and relatively simple to test exhaustively; sensor software, again well understood and amenable to extensive testing; and the bulk of the software of initial and continuous built-in tests for sensors and actuators.

In addition to the Ada verification there are twenty-seven low-level assembler routines that are called numerous times in the Ada. These amount to about 10,000 lines of assembler with floating point calculations. These assembler routines have been verified using the Malpas tool [253] against specifications derived from the Ada verification.

About ten mismatches were found between the specification and the implementation produced from an incremental development process that has been in action for ten years.

7.4 Concurrency Verification

There are approximately 250 scheduled elements on the three processors on which the control laws execute. The use of the FDR refinement model checker allows the verification to be completely automated as described in Section 6.6; however, the amount of concurrency permitted by the Simulink specification required a hierarchical verification to be developed to address the state-space problems.

The hierarchical approach checks a number of subsystems in the diagram hierarchy, hiding that part of the scheduling that does not pertain. A new subsystem is formed from those subsystems below it in the hierarchy, and the verification process repeated. This approach has also allowed an incremental verification so that the process has not been held up by the incremental delivery of the specification.

The verification check is not yet complete because the final part of the Simulink specification has not yet been delivered. It is estimated that the total verification of the concurrency will be about four person-months. Currently about ten mismatches have been found. These cover:

- Interactions in the code that are not in the specification.
- Over-scheduling.
- Scheduling that violates the causality in the diagram.

The first category of anomaly means that a non-local variable is written to, and apparently used, in another procedure. The second category is when a procedure is called more times than required by the specification; this might have no impact except for wasting execution cycles. The final category is more serious, an example of this category in terms of Figure 6 would be if the subsystem ControlE was executed before the subsystem ControlMode. Clearly the causality implied by the diagram in Figure 6 is violated because one of the inputs to ControlE is calculated by the subsystem ControlMode.

7.5 Exception Analysis

The Malporte tool has been used to assess all 180,000 lines of Ada. To pass all the code through Malporte took about six weeks, requiring about two person-

months of effort. This produced approximately 1,500 anomalies. In terms of actual processing by Malporte, it took 25 minutes to analyse 278 files using a 266MHz Pentium 2 processor.

The analysis is pessimistic, therefore most of these anomalies could be resolved after further information is received. For example, it is known that the type used for variables receiving sensor information has a much greater range than the sensor could ever supply. Restricting the range to a narrower set of values would probably remove a great number of potential overflow anomalies reported by Malporte. Details of the use of the Malporte tool for other procurements can be found in [252].

7.6 Worst-Case Execution Time

A specific Worst Case Execution Time, WCET, analysis tool was developed 10 years ago by the developer of the safety critical system being currently assessed for acceptance. The tool has been used extensively on the project and assessed, therefore it was not considered cost effective to repeat WCET analysis.

The concurrency verification described earlier crucially relies on WCET analysis. The CSP model of the implementation assumes that scheduled procedures complete before the end of a fixed timed segment. The end of this fixed timed segment is represented by a synchronisation on the *tock* event within the CSP model of the implementation.

8 Summary

A set of tools have been developed, as well as using existing tools, and integrated into a framework for accepting safety critical systems. This framework, and the tools and techniques within it, have been used for the acceptance of a significant hard real-time safety critical system.

The WCET tool is based on a new symbolic execution algorithm that is less pessimistic than the traditional WCET algorithm. The tool is based on Java Byte Code and is easily characterized for different languages, compilers, processors and configurations. WCET analysis is an essential component in gaining assurance for future time-critical real-time systems. Many such control systems are to be found in new and legacy military systems.

Well-defined acceptance techniques can be used to show whether contractual conditions have been met; for example, that the safety assurance is adequate. This should lead to developers incorporating it into their development processes. The reason for this is that a developer will wish to ensure that it will be paid for achieving its contractual obligations. The acceptance techniques should ideally be designed to be independent of the development process in order for them to be as widely applicable as possible. In practice, it has been shown that some knowledge of the development process has to be taken into account in order to join them up cost-effectively.

The adoption of acceptance techniques by the developer would also lead to more effective testing and cost reductions, because there would be less re-work due to the cycle of fixing bugs, testing and then fixing them again.

8.1 Relationship with Other Techniques

As part of the FORWARD project on behalf of the UK Department of Trade and Industry, the Systems Assurance Group have investigated basic mechanisms for assuring quality of service in *ad hoc* networks. These *ad hoc* networks are a core component of future ubiquitous computing, environments. CSP_M and its associated model checking tool, FDR, was used to address clear-cut questions of correctness for routing and distributed data replication respectively. In many cases, it is not possible to guarantee that a protocol achieves its goals with absolute logical certainty; but it can be argued that the counterexamples will arise with negligible (or zero) probability.

As part of the FORWARD project performance characteristics were determined by calculating accurate minimum and maximum bounds on the probability of a specified condition being satisfied. The methodology makes use of Birmingham University's PRISM model checker [210] and a translator from a probabilistic version of CSP_M, called pCSPM [101], to PRISM developed within FORWARD. The methodology includes guidance on how to extend possibilistic models written in CSP_{MP} into probabilistic models (written in pCSPM) suitable for performance analysis. The methodology and tools were applied to part of the GRID protocol suite.

As described in this chapter, control systems using a simple distributed cyclic scheduling have been verified using a combination of representations in CSP_M and specification statements using Z. Subsequently the *Circus* language has been used to combine concurrent and sequential verification and reported in [47]. More sophisticated schedulers use probabilistic scheduling to maximize the efficiency of the computing resources. Probabilistic software protocols are also used to tolerate random faults that can occur on the computing platform and infrastructure. The growing pervasiveness and sophistication of computing devices and their internet working present daunting challenges if society is to depend upon them. These types of systems of systems require methods based on theories and tools that combine refinement, probabilistic reasoning and concurrency. The body of work presented in this book is a significant step in meeting this growing challenge.

Techniques for Temporal Logic Model Checking

David Déharbe

Universidade Federal do Rio Grande do Norte
Departments de Informática e Matemática Aplicada
Natal - RN, Brazil

Model checking is a set of formal verification techniques that aim to show that a structure representing a computational system (for instance, a protocol, or a hardware or a software component, among others) is a model for a property that represents a requirement for this system. Many model-checking approaches have been proposed, depending on the formalism the property is expressed in, and the class of structures used to represent the system under verification.

In the next section, we motivate the use of model-checking, and summarize the research work that has been carried out in this area. In Section 2, we discuss Kripke structures, one of the main state-transition models in the model-checking literature. Section 3 is devoted to propositional temporal logics commonly used in the realm of program verification: CTL*, CTL and LTL. Their expressiveness is compared and illustrated. Section 4 provides the key components of explicit model-checking for the temporal logic LTL, and introduces some advanced techniques that are used in current implementations such as the SPIN model-checker. Then, in Section 5, we present the decision procedure for CTL as originally proposed in the seminal work of [62] and [218]. Next, in Section 6, we present how Kripke structures can be represented and analyzed using quantified propositional logic, and introduce Binary Decision Diagrams (BDDs), an efficient implementation of this logic; at that point we are ready to present symbolic model-checking. We also present the symbolic model-checking features of the NuSMV model-checker. Section 7 is devoted to so-called bounded model-checking. This technique makes it possible to verify (in general partially) LTL properties of much larger structures than the previous approaches. We also present the bounded model-checking features of NuSMV. Finally, in Section 8, we briefly touch upon some of the most recent developments in formal verification of software, some of which employ techniques that had been initially developed to enhance model checkers.

1 Introduction

The last decades have seen computation devices getting smaller, cheaper and requiring less power. Therefore, computers are getting more and more pervasive in our world, and an ever increasing number of critical activities are being supervised and controlled by a combination of software and hardware. Engineering computation devices in a timely fashion, however, is a difficult and error prone task, as witnessed by the struggle in which the software engineering commu-

A. Cavalcanti, A. Sampaio, and J. Woodcock (Eds.): PSSE 2004, LNCS 3167, pp. 315–367, 2006.

nity has been engaged to provide the tools and methodology that the software industry needs to create robust products.

Following the tradition of other engineering disciplines, a branch of computer science has strived to propose methods based on sound principles drawn from mathematics: *formal methods*. An example of a software design methodology based on formal methods is the B method [3]. It commends that an initial, high-level specification of the system be developed, possibly reusing existing specifications of components. This specification is a mathematical description of the requirements of the system and is expressed in a suitable language (in this case, the B notation). The specification needs to be analyzed in detail to verify that it satisfies the requirements of the project, and that it does not contain internal inconsistencies. Of course this analysis should have a sound basis.

Once the specification has been analyzed, it can then be refined, reflecting implementation and design choices, towards a more detailed model of our intended view of the system. Again, when a refinement is performed, it is important to guarantee that it has preserved the initial intentions of the designers; this may require further analysis. At some point, if the model is sufficiently detailed, it can be mapped to a computational model, such as code in a programming language. Of course, it is also necessary to show that this mapping does not introduce unexpected behaviours. In summary, specification, refinement and code synthesis all require some kind of analysis. The point here is that, as we have a specification language with well-defined semantics, we can use formal verification techniques, that is, mathematical reasoning, to carry out the analysis.

Formal verification, however, is not restricted to design methodologies that rely heavily on formal methods and refinement as suggested above. The implementation of a design can still be verified if we can map it to some mathematical structure that can support some proof activity. For instance, formal verification techniques have been directly applied on an industrial basis to software to detect errors in device drivers, or to show the absence of run-time errors in flight-control software [30]. A common trait to all these formal verification activities is that they suppose that a mathematical structure represents (or models) a computing system, and all the verification activity is carried out on the mathematical structure. Thus, formal verification is used to show properties of some (abstract) model of the system instead of the system itself.

Although it is theoretically possible to carry out verification tasks by hand, the sheer size of the problems encountered in practice makes it impossible to use any but a computer-aided verification technique. Thus, researchers have been busy developing a very large number of formal verification techniques addressing different classes of mathematical structures and specification logics. These approaches are generally classified as either theorem provers or model checkers. In general, the richer the mathematical structure and the more expressive the specification logic, the harder gets the verification. For software in general, the problem is not even decidable. So although theorem provers (like, for example, PVS, Coq, ACL2, Simplify, Mona, and haRVey) have a much broader scope, they may require manual intervention and considerable human expertise.

Model checking is a decision procedure to check that a given structure is a model of a given formula. For instance, in the setting of propositional logic, model checking amounts to the problem of deciding if a formula is satisfiable in a given model, that is, a boolean assignment of the variables of this formula. In the early 80s, [62] and [218] presented independently a fully-automatic model-checking algorithm for formulas of the branching-time temporal logic CTL, and finite-state transition systems called Kripke structures. In this seminal work, the verification is performed as a depth-first graph traversal, of complexity linear in both the size of the formula and in the size of the model. This algorithm has been used to verify systems of up to several million states and transitions, which is enough in practice only for very small systems. For example, a system composed of four concurrent processes, each having three variables that can take five different values, has 250 million states.

In order to provide a model-checking system of practical interest for the industry, it was therefore necessary to go beyond this initial approach. In the past twenty years, researchers have been very active and successful to provide solutions to improve the situation. Also, the temporal logic CTL (Computation Tree Logic) and the Kripke structure computation model has some limitations, and there has been a lot of effort towards the definition of model-checking approaches for other models of computation, as well as other specification logics.

Automata-based specification and verification was first advocated in [249] and further developed by several researchers including [141] and [120]. In this approach, desired (or undesired) behaviours are described as Büchi automata, which are automata that recognize infinite words, and are, therefore, suitable to specify reactive systems. It has been further shown that temporal logic LTL (Linear Time Logic) formulae can be expressed as Büchi automata [99]. This framework was used as the basis for further developments on so-called explicit model-checking, where the states of the Kripke structures are explicitly enumerated and the state space is visited in a depth-first search. This framework was enriched using partial-order reduction [211], state-compression techniques [119], and bit-state hashing [118], resulting in very mature formal verification tools such as SPIN [120].

Another major line of research has been initiated in [185], where it is proposed to represent the state-transition graphs and the search algorithms as boolean logic operations that can be efficiently implemented with binary decision diagrams (BDDs) [34]. In this approach, called symbolic model-checking, BDDs are used to represent and operate on the characteristic functions of both sets of transitions and sets of states of the graph. Since sets are not explicitly enumerated, but represented by their characteristic functions, the size of the verified model is not necessarily bound by the memory of the computer carrying out the verification. This opens the possibility to verify systems that are several orders of magnitude larger than was previously achieved. However, in practice, BDDs do not cope efficiently with formulas with more than a few hundred variables and the exponential growth of the size of the state space to be explored

makes it impractical to directly apply symbolic model-checking on large or even medium-size industrial examples.

More recently, a new symbolic model-checking approach, based on extremely efficient satisfiability solvers for propositional logic, has yielded very interesting results. These new techniques, known as bounded model-checking, can explore much larger Kripke structures than BDD-based approaches, however they are limited in the depth of the search in the state-space. They are, therefore, most useful to detect bugs that can be exhibited in relatively short paths from some initial state of the structure.

In this chapter, we provide an overview of the most well-known techniques of model checking. We also compare this more traditional line of work with the approach called refinement model-checking adopted in the tool FDR presented in Chapter 4.

2 Kripke Structures

Among the numerous concurrency models proposed in the last 20 years [254], Kripke structures are one of the most commonly used in the scope of temporal-logic model-checking. Kripke structures are interleaving, nondeterministic, state-based models suitable to represent a wide-range of practical problems, for instance hardware, protocols and concurrent software.

2.1 Definitions

Informally, Kripke structures are finite-state transition systems, where the labelling is associated with states, instead of the more traditional labelling of transitions.

Definition 1 (Kripke structure). *Let P be a finite set of boolean propositions. A Kripke structure over P is a quadruple $M = (S, T, I, L)$ where:*

- *S is a set of states (when S is finite, we say that M is a* finite Kripke structure*);*
- *$T \subseteq S \times S$ is a transition relation, such that $\forall\, s : S \bullet \exists\, s' : S \bullet (s, s') \in T$;*
- *$I \subseteq S$ is the set of initial states;*
- *$L : S \to \mathbb{P}\, P$ is a labelling function.*

The labelling function L associates each state with a set of boolean propositions true in that state.

Example 1 (Kripke structure). In the alternating-bit protocol, the sender tags messages with a control bit set to high and low in alternation. The receiver acknowledges each message attaching the corresponding control bit. The sender waits for the acknowledgement and checks that the control bit is correct before sending another data message. If necessary, the same data is sent again.

The Kripke structure *ABPsender* below describes the sender component in the alternating-bit protocol. The set of atomic propositions is $P = \{\, b, g, s, w \,\}$.

The proposition b indicates the state of the control bit; g is set when the sender is getting data from the user; s states that the sender is sending the message to the transmission medium; finally, w indicates that the sender is waiting for the acknowledgement. The data messages are not modelled in this version of $ABPsender$. It is defined by the structure (S_A, T_A, I_A, L_A) where:

- the set of states is $S_A = \{ s_0, s_1, s_2, s_3, s_4, s_5 \}$;
- the transition relation is

$$T_A = \{ (s_0, s_0), (s_0, s_1), (s_1, s_2), (s_2, s_1), (s_2, s_3), (s_3, s_3), (s_3, s_4), (s_4, s_5), (s_5, s_4), (s_5, s_0) \}$$

- the set of initial states is $I_A = \{ s_0, s_3 \}$;
- the labelling function is

$$L_A = \{ s_0 \mapsto \{g\}, s_1 \mapsto \{s\}, s_2 \mapsto \{w\}, s_3 \mapsto \{g, b\}, s_4 \mapsto \{s, b\}, s_5 \mapsto \{w, b\} \}$$

The state transition diagram of $ABPsender$ is displayed in Figure 1.

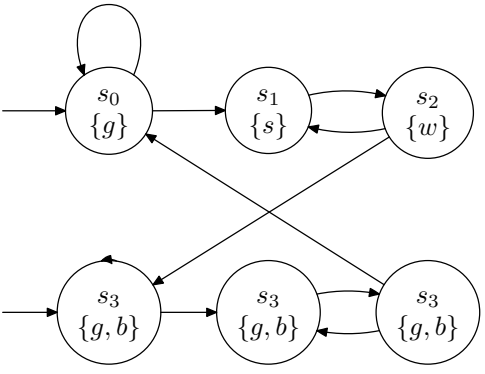


Fig. 1. State transition diagram for Kripke structure $ABPsender$

Initially, the sender is in one of two states: s_0 and s_3, modeling the fact that it expects a message to be delivered (g is true) and the control bit b can be either true (state s_3) or false (state s_0). For sake of conciseness, we explain the behaviour from state s_0. The sender remains in state s_0 until it gets a message and goes to state s_1, representing the delivery of the message tagged with the bit 0. It then goes to state s_2, modeling the situation where the sender waits the acknowledgement. If the acknowledgement is not as expected (it may be wrongly tagged or there might be some time-out), then the sender needs to send the tagged message again, which is represented by the transition to state s_1. Otherwise, the acknowledgement may come as expected, and the sender can

deliver a new message, with the opposite tag. This is modelled by the transition to state s_3. The possible behaviours from s_3 are the same as those from s_0, with the tag now set to 1.

A path π in the Kripke structure M is a possibly infinite sequence of states $(s_1, s_2, \ldots)$ such that $\forall\, i \mid i \geq 1 \bullet (s_i, s_{i+1}) \in T$. The i-th element $\pi(i)$ in a path π is the i-th state of π. The set RS of states reachable from a set of states I contains the states to which there is a path starting from a state in I:

$$RS = \{\, s : S \mid \exists\, \pi \bullet \pi(1) \in I \wedge \exists\, i \mid i \geq 1 \bullet \pi(i) = s \,\} \tag{1}$$

We say that a state label l is reachable, if there is a reachable state labelled l.

In our example Kripke structure, all states are reachable, since the state transition graph is connected, but many state labels are unreachable. For instance, no reachable state is labelled $\{\, g, w \,\}$. In general, if there are p different propositions, there are 2^p possible state labels. In most cases, the number of reachable state labels is only a fragment of the number of possible labels.

Exercise 1. Define a Kripke structure *ABPreceiver* that models a high-level behaviour of the receiver in the alternating bit protocol. An informal description of the receiver is that it waits for tagged messages coming from the transmission medium. If the message has the expected tag, then it is forwarded to the upper level client in the transmission protocol, and a tagged acknowledgement is sent back to the transmission medium. To make the exercise simple, do not model the message and the acknowledgement, only the different states the component can be, much as in the same manner as the sender.

Draw a diagram of the transition graph, and identify explicitly the states, transitions, initial states and labels of *ABPreceiver*.

2.2 Computation Tree

Computation trees are a special class of Kripke structures in which the transition relation is acyclic, and only branches away from the initial states. The computation tree of a Kripke structure is obtained by an operation similar to that of unfolding in CCS [188].

Definition 2 (Computation tree). *Let P be a finite set of boolean propositions. A computation tree is a Kripke structure $M = (S, T, I, L)$ over P, such that:*

- *every state is reachable;*
- *$\forall\, s : S \bullet (s, s) \notin T^+$, where T^+ is the transitive closure of T;*
- *$\forall\, s, s', s'' : S \bullet (s', s) \in T \wedge (s'', s) \in T \Rightarrow s' = s''$.*

For any Kripke structure M, it is possible to associate a computation tree M' such that the set of states of M' is isomorphic to the set of finite paths of M.

Definition 3. *Let $M = (S, T, I, L)$ be a Kripke structure. The computation tree of M, denoted $ct(M)$, is the Kripke structure (S', T', I', L') such that:*

- S' *consists of all finite paths of* M *that start at initial states;*
- $(\pi, \pi') \in T'$ *iff* $\pi = (s_1, \ldots s_n)$, $\pi' = (s_1, \ldots, s_n, s_{n+1})$ *and* $(s_n, s_{n+1}) \in T$*;*
- I' *consists of all paths of* M *with only one (initial) state;*
- *For any path* $\pi = (s_1, \ldots s_n)$ *of* M, $L'(\pi) = L(s_n)$.

Example 2 (Computation tree). Figure 2 depicts the initial part of the state transition diagram of the infinite Kripke structure $ct(ABPsender)$. For instance, state π_9 in $ct(ABPsender)$ corresponds to path $s_3 s_4 s_5$ in $ABPsender$. Note how the labelling function is preserved between the origin and destination of the transitions in the original Kripke structure and the corresponding computation tree.

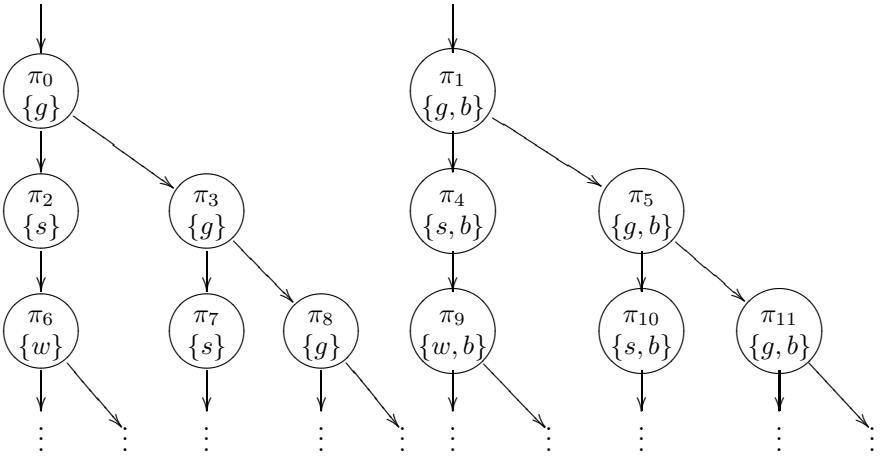


Fig. 2. Partial state transition diagram of $ct(ABPsender)$

Exercise 2. Draw the initial part of the state transition diagram of the Kripke-structure $ct(ABPreceiver)$, which is partially presented in Figure 2.

3 Temporal Logics

Temporal logics are formal systems that were developed by logicians and philosophers to describe and reason about the evolution of the truth of propositions. In addition to the classic logical operators they include temporal operators that make it possible to assert that a proposition may become true or will eventually become true. Pnueli [214] was the first to observe that temporal logics can be used to specify and analyse programs that run continuously, such as those in embedded software or operating systems. For instance, in a resource scheduler, we may want to verify that all processes eventually get access to the resource.

There is an impressive body of work on the use of temporal logics in computer science in general, and in program specification and verification more specifically. The interested reader is referred to [84] for a survey on research in this area, and the main available results. Here, we first introduce a very simple temporal logic, ω-regular expressions, and then focus our attention on qualitative propositional temporal logics that have been successfully employed in the realm of model checking: CTL* and its subsets CTL and LTL.

3.1 ω-Regular Expressions

Regular expressions are a well-known language to denote sets of words. The operators are ϵ (empty word), . (concatenation), + (choice), $*$ (repetition). If we consider as alphabet the set of states of a Kripke structure, a word is a sequence of states. For instance, for $ABPsender$, the regular expression $s_0(s_1\,s_2)^+s_3$ represents all the sequences of states that initiate in s_0, go through s_1 and s_2 a finite, non-null number of times, and then end in s_3. The + operator represents non-empty repetition.

In our context, paths can be viewed as infinite words, and regular expressions fall short of expressiveness as they only allow to represent sets of finite words. The ω-regular expressions extend this notation with the ω operator that stands for infinite repetition. For instance, the ω-regular expression $s_0(s_1\,s_2)^\omega$ represents the unique path where the $ABPsender$ initiates in s_0 and then loops infinitely over the states s_0 and s_1.

The ω-regular expressions provide constructions to express sets of paths and provide a linear vision of behaviour. They are thus classified as a *linear temporal logic*. However ω-regular expressions are a bit awkward to read and write and do not serve well as a specification formalism.

Exercise 3. Use an ω-regular expression to denote some paths of the Kripke structure $ABPreceiver$.

3.2 The Logic CTL*

Temporal logics may be classified according to various criteria. Is the time modelled as a continuous or discrete quantity? Is the behaviour considered as a (possibly infinite) linear set of executions, or as a branching tree? Is the underlying logic propositional or first-order? Are we interested in considering both the past and the future? The combined answer to such questions determines what kind of temporal logic is needed. Of course, computer scientists need to be pragmatic, and having an efficient algorithmic approach to reason about such logic is also of paramount importance.

The computation tree logic CTL* makes it possible to describe and reason about the behaviour of Kripke structures. It has the following characteristics: the time is discrete; the underlying logic is propositional; and it has modalities about

the future (but not about the past). Moreover, CTL* accepts both the branching and linear views of the temporal succession of states. Finally, there are (relatively) efficient algorithms for two important sublogics of CTL*, namely CTL and LTL.

In addition to the usual logic connectives, CTL* contains two types of modalities: path quantifiers and linear time operators, that can also be viewed as state quantifiers. There are two path quantifiers, which are described below.

- **E**, the existential path quantifier: $\mathbf{E}\phi$ holds in state s if ϕ holds on some path leaving s;
- **A**, the universal path quantifier: $\mathbf{A}\phi$ holds in state s if ϕ holds on all paths leaving s.

The linear time operators are as follows.

- **X**, on the next state: $\mathbf{X}\phi$ holds on path π if ϕ holds in the second state of π;
- **F**, on some future state (or eventually): $\mathbf{F}\phi$ holds on path π if ϕ holds in at least one state of π;
- **G**, on all future states (or globally): $\mathbf{G}\phi$ holds on path π if ϕ holds in at all states of π;
- **U**, the until operator: $\phi\mathbf{U}\psi$ holds on path π if ψ holds in some state s of π and ϕ holds on all states of π preceding s;
- **W**, the weak until operator: $\phi\mathbf{W}\psi$ holds on path π if ϕ holds in every state of π up to the first state that satisfies ψ.

Syntax. The syntax of CTL* is defined as follows. Given a set of atomic propositions AP, the set of state formulae and the set of path formulae are the least sets such that:

- if $p \in AP$ then p is a state formula;
- if f and g are state formulae, then so are $\neg f$ and $f \vee g$;
- if f is a path formula, then $\mathbf{E}f$ and $\mathbf{A}f$ are state formulae;
- if f is a state formula, then f is also a path formula;
- if f and g are path formulae, then $\neg f$, $f \vee g$, $\mathbf{X}f$, $\mathbf{G}f$, $\mathbf{F}f$, $f\mathbf{U}g$ and $f\mathbf{W}g$ are path formulae.

Semantics. The semantics of a CTL* formula over the set of atomic propositions AP is defined in terms of an underlying Kripke structure on the same set of atomic propositions AP. If ϕ is a state formula, the notation $M, s \models \phi$ represents the fact that ϕ is valid in the state s of the Kripke structure M. Similarly, if ψ is a path formula, the notation $M, \pi \models \psi$ represents the fact that ψ is valid in the path π of the Kripke structure M.

In the following, we use p to denote an arbitrary atomic proposition, ϕ_1 and ϕ_2 to denote state formulae, and ψ to denote path formulae. The relation $M, s \models \phi$ is defined inductively as follows.

$$
\begin{array}{lcl}
M, s \models p & \Leftrightarrow & p \in L(s) \\
M, s \models \neg\phi_1 & \Leftrightarrow & M, s \not\models \phi_1 \\
M, s \models \phi_1 \vee \phi_2 & \Leftrightarrow & M, s \models \phi_1 \text{ or } M, s \models \phi_2 \\
M, s \models \mathbf{E}\psi & \Leftrightarrow & \exists\, \pi \bullet \pi(1) = s \wedge M, \pi \models \psi \\
M, \pi \models \phi & \Leftrightarrow & M, \pi(1) \models \phi \\
M, \pi \models \neg\psi & \Leftrightarrow & M, \pi \not\models \psi \\
M, \pi \models \psi_1 \vee \psi_2 & \Leftrightarrow & M, \pi \models \psi_1 \text{ or } \pi \models \psi_2 \\
M, \pi \models \mathbf{X}\psi & \Leftrightarrow & M, \pi(1) \models \psi \\
M, \pi \models \psi_1 \mathbf{U} \psi_2 & \Leftrightarrow & \exists\, k \mid k \geq 1 \bullet M, \pi(k) \models \psi_2 \wedge \\
 & & \forall\, j \mid 1 \leq i < k \bullet M, \pi(j) \models \psi_2
\end{array}
$$

The semantics of the other boolean operators (such as $\wedge$, $\Rightarrow$, $\Leftrightarrow$) is defined as usual. The semantics of the remaining temporal modalities is defined by the following rules.

$$
\begin{array}{c}
\mathbf{A}\psi = \neg\mathbf{E}\neg\psi \\
\mathbf{F}\psi = true\mathbf{U}\psi \\
\mathbf{G}\psi = \neg\mathbf{F}\neg\psi \\
\psi_1 \mathbf{W} \psi_2 = (\psi_1 \mathbf{U} \psi_2) \vee (\mathbf{G}\psi_1)
\end{array}
$$

The Logic CTL. Computation Tree Logic (CTL for short) is a subset of CTL* that is restricted to reason about only the branching nature of program execution. Therefore, it only allows branching operators that are composed of a path quantifier immediately followed by a state quantifier. These operators are **EX**, **AX**, **EF**, **AF**, **EG**, **AG**, **E**[**U**], **A**[**U**], **E**[**W**], **A**[**W**]. As for CTL*, the semantics of CTL formulae is defined with respect to the states of a Kripke structure, and we adopt the same notation $M, s \models \phi$ to represent that the CTL formula ϕ is valid in the state s of M.

Definition 4. *A formula f is valid in structure M if it is valid for all initial states:*

$$M \models f \text{ iff} \forall\, s : I \bullet M, s \models f.$$

Note that the full expressiveness of CTL with respect to time can be obtained with the operators **EX**, **EF** and **E**[**U**]:

$$
\begin{array}{ll}
\mathbf{AX}f = \neg\mathbf{EX}\neg f & \mathbf{A}[f\mathbf{U}g] = \neg\mathbf{E}[\neg g\mathbf{U}\neg f \wedge \neg g] \wedge \neg\mathbf{EG}\neg g \\
\mathbf{AG}f = \neg\mathbf{EF}\neg f & \mathbf{E}[f\mathbf{W}g] = \neg\mathbf{A}[\neg g\mathbf{U}\neg f \wedge \neg g] \\
 & \\
\mathbf{AF}f = \neg\mathbf{EG}\neg f & \mathbf{A}[f\mathbf{W}g] = \neg\mathbf{E}[\neg g\mathbf{U}\neg f \wedge \neg g] \\
\mathbf{EF}f = \mathbf{E}[true\mathbf{U}f] &
\end{array}
$$

Figure 3 pictures the semantics of the four universal temporal operators (vertical dots indicate paths where f holds infinitely).

Example 3 (CTL formulae). The following CTL formulae state properties of the *ABPsender* Kripke structure:

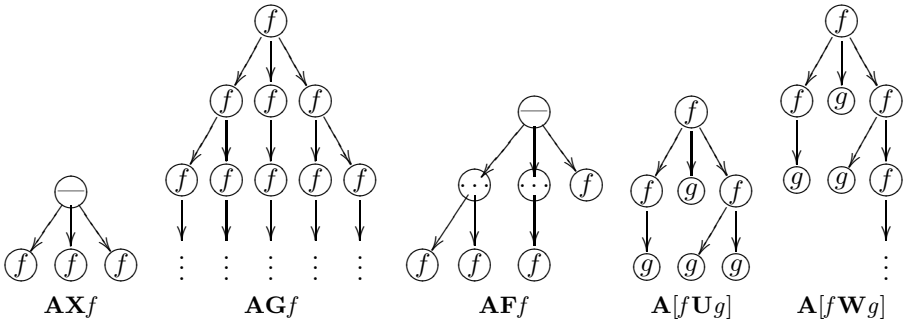


Fig. 3. Illustration of universal CTL operators

- $\mathbf{AG}((s \wedge \neg w \wedge \neg g) \vee (\neg s \wedge w \wedge \neg g) \vee (\neg s \wedge \neg w \wedge g))$: the sender is always getting from the sender, sending a message, or waiting for an acknowledgement.
- $\mathbf{EG}(\neg s \wedge \neg w)$: there is an execution path of the sender such that it is never sending or waiting.
- $\mathbf{AG}(w \wedge b \Rightarrow \mathbf{A}[(w \wedge b)\mathbf{W}((s \wedge b) \vee (g \wedge \neg b))])$: the sender keeps waiting for an acknowledgement with the tag bit set, until it either goes to the sending state again with the same tag or goes to get a message from the client with the bit tag unset.
- $\mathbf{AGEF}g$: the sender can always go back to a state where it can process requests from the user.

Exercise 4. Write CTL properties that you expect *ABPreceiver* to satisfy. Write CTL properties that you expect *ABPreceiver not* to satisfy.

The Logic LTL. Linear Temporal Logic (or LTL for short) offers a linear viewpoint on the progress of the Kripke structure behaviour. LTL formulae have the form $\mathbf{A}\phi$, where ϕ is a path formula that does not contain further path quantifiers, so that the only state sub-formulae are atomic propositions. It is customary to drop the initial universal path quantification when writing LTL formulae. So, basically, the temporal logic operators of LTL are $\mathbf{X}$ (also denoted $\bigcirc$), $\mathbf{F}$ (also denoted $\Diamond$), $\mathbf{G}$ (also denoted $\Box$), $\mathbf{U}$ (also denoted $\mathcal{U}$) and $\mathbf{W}$ (also denoted $\mathcal{W}$); they retain their usual semantics from CTL*. For instance, the LTL formula $\mathbf{F}\,\mathbf{G}\mathit{initialized}$ states that on all execution paths, from some point on, the proposition *initialized* is always true.

Expressiveness. The expressive power of CTL and LTL cannot be directly compared. For example, to see that CTL may express properties that cannot be expressed in LTL, consider the formula $\mathbf{AGEF}\phi$: in every state, there is path that leads to a state where ϕ holds. As LTL does not have path quantifiers, one cannot express this property: indeed, the candidate formula $\mathbf{GF}\phi$ does not match as it express that, on every path, ϕ will eventually hold (see Figure 4).

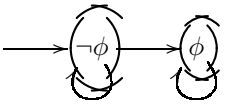


Fig. 4. Kripke structure where **AGEF**ϕ holds, but where **GF**ϕ does not hold

Conversely, the LTL formula **FG**ϕ has no equivalent CTL formula [60]. Indeed, as already explained, this formula states that, on every path, there is a point after which ϕ is always true. In CTL, we would use the **AF** operator to state that from some point on ϕ is always true. However, as the argument of **AF** we would use either an existential path operator or a universal path operator, and therefore would not be asserting anything about the *current* path, but about *some* path or *all* paths from a certain point in the future (see Figure 5).

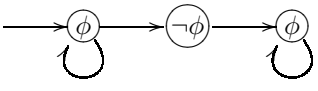


Fig. 5. Kripke structure where **AFAG**ϕ does not hold, but where **FG**ϕ holds

Finally, CTL* is strictly more expressive than the logics LTL and CTL, as all LTL and CTL formulas are also CTL* formulas. Thus, combining our two previous examples, the CTL* formula **FG**$\phi \vee$ **AGEF**ϕ can be expressed neither in LTL nor in CTL.

Comparison with Approaches Based on Process Algebra. The approach to model checking that we describe assumes that the specification is given in a temporal logic and the implementation is a kind of labelled transition system (LTS). Proponents of process algebra such as CSP have a different approach: both the specification and the implementation are expressed as processes.

A CSP process P has an associated labelled transition system $M(P)$ that characterizes its operations semantics. Labelled transition systems are similar to Kripke structures, however, their labels are carried by transitions. An LTS $M(P)$ has an associated set of *traces* $T(P)$ (conceptually, this corresponds to the set of paths of a Kripke structure). An implementation process P_i is correct with respect to a specification process P_s if $L(P_i) \subseteq L(P_s)$. This correctness relation is called a *trace refinement* relation. Other, more elaborated trace-like semantics for CSP have been proposed and are the basis of the FDR verification tool.

Note that while trace refinement preserves validity of LTL formulae, this is not the case of CTL, as this logic allows existential path quantification. First, consider the case of LTL. An LTL formula states a property that all paths must comply.

Moreover, LTL formulas do not allow to state properties about branching, as nested path quantifiers are disallowed. Thus an LTL formula φ can be viewed as a linear pattern that a path (trace) π may or may not satisfy, which is denoted by $\pi \models \varphi$. So, basically, stating that a process P_s satisfies an LTL formula φ means that $\forall \pi \in T(P_s) \bullet \pi \models \varphi$. Now considering that refinement restricts the set of possible paths, it is easy to see that if P_i refines P, then $\forall \pi \in T(P_i) \bullet \pi \models \varphi$, and P_i satisfies φ.

Now, consider the case of a CTL formula, such as **AGEF**ϕ. Consider a process P with an LTS such as that in Figure 4; clearly, $P \models$ **AGEF**ϕ. A possible trace refinement P_i of P would be to remove the transition to the state labeled with ϕ, and the resulting LTS would no longer be a model for our example formula.

3.3 Property Patterns

It is a bit tricky to express correctly properties of interest in temporal logics such as LTL and CTL. Too often, the inexperienced writer of temporal logic formulae fails to convey the intended meaning and may be fooled by the verification results obtained. To avoid such counterproductive errors, repositories of property patterns have been developed [175].

We present some of the simplest (and fortunately also the most common) patterns that have thus been established, showing their intended meaning and their corresponding expression in CTL and LTL. [175] defines patterns of properties, parameterized by some conditions which we denote ϕ, ψ, ψ_i. One can also view *phi*, ψ, ψ_i as events, and the patterns refer to occurrences of these events. These patterns are organized as an orthogonal combination of a hierarchy of property and scope patterns. At the top-level, property patterns are classified as either occurrence patterns or order patterns. Occurrence patterns are further divided into:

absence(ϕ) the event ϕ does not occur;
universality(ϕ) the event ϕ always occur;
existence(ϕ) the event ϕ occurs at least once;
bounded existence(ϕ, k) the event ϕ has a fixed, maximum number k of occurrences.

Order patterns are themselves divided into:

precedence(ϕ,ψ) event ϕ occurs (strictly) before event ψ;
response(ϕ,ψ) after ϕ, then ψ necessarily occurs;
precedence chain(ϕ,ψ_1,ψ_2) event ϕ is always preceded by events ψ_1 and ψ_2 (in that order);
response chain(ϕ,ψ_1,ψ_2) if ψ_1 and ψ_2 occur (in that order), then necessarily ϕ will occur;
constrained chain(ϕ,ψ_1,ψ_2,ψ_3) if ϕ occurs, then ψ_1 and ψ_2 occur in that order, and ψ_3 shall not occur.

Finally, the scope patterns are:

global(ϕ) ϕ holds globally;
before(ϕ, ψ) ϕ holds before some event ψ;
after(ϕ, ψ) ϕ holds after some event ψ;
between(ϕ, ψ_1,ψ_2) ϕ holds between two events ψ_1 and ψ_2;
until(ϕ, ψ_1,ψ_2) ϕ holds after event ψ_1 and until ψ_2 happens.

Table 1 presents the CTL and LTL expressions of the absence property patterns for the different possible scopes, while Table 2 contains the CTL and LTL formulas of the nine different property patterns for the global scope (see [175] for a complete presentation of the patterns).

In Table 1, the first pattern state the global absence of ϕ; the second pattern states the absence of ϕ before ψ; the third pattern states the absence of ϕ once ψ has occurred; the fourth states that, between occurrences of ψ_1 and ψ_2, ϕ does not occur; the fifth pattern states that, once ψ_1 has occurred, ϕ does not occur until ψ_2 occurs. Note the subtle difference between the fourth and fifth patterns. Only the fifth pattern expresses the absence of ϕ when ψ_1 occurs but is not eventually followed by ψ_2.

Table 1. Absence property patterns (ϕ is the place-holder for the absent event)

scope	logic	pattern
global	CTL	$\mathbf{AG}\neg\phi$
	LTL	$\Box\neg\phi$
before ψ	CTL	$\mathbf{A}[(\neg\phi \vee \mathbf{AG}\neg\psi)\mathbf{W}\psi]$
	LTL	$(\Diamond\psi) \Rightarrow (\neg\phi\,\mathcal{U}\,\psi)$
after ψ	CTL	$\mathbf{AG}(\psi \Rightarrow \mathbf{AG}\neg\phi)$
	LTL	$\Box(\psi \Rightarrow \Box\neg\phi)$
between ψ_1 and ψ_2	CTL	$\mathbf{AG}((\psi_1 \wedge \neg\psi_2) \Rightarrow \mathbf{A}[(\neg\phi \wedge \mathbf{AG}\neg\psi_2)\mathbf{W}\psi_2])$
	LTL	$\Box((\psi_1 \wedge \neg\psi_2 \wedge \Diamond\psi_2) \Rightarrow (\neg\phi\,\mathcal{U}\,\psi_2))$
after ψ_1 until ψ_2	CTL	$\mathbf{AG}(\psi_1 \wedge \neg\psi_2 \Rightarrow \mathbf{A}[\neg\phi\mathbf{W}\psi_2])$
	LTL	$\Box((\psi_1 \wedge \neg\psi_2 \Rightarrow (\neg\phi\,\mathcal{W}\,\psi_2))$

Table 2 shows the CTL and LTL patterns of properties with a global scope, that is, holding forever, and is divided into two parts containing respectively four occurrence and five order patterns. In the first part of the table, the first pattern expresses global absence of ϕ, the second pattern expresses that ϕ always remains valid, the third pattern states that ϕ occurs at least once, and the fourth pattern states that ϕ occurs at most twice. In the second part, the fifth pattern states that (the first occurrence of) ψ occurs before (the first occurrence of) ϕ, the sixth pattern states that every occurrence of ϕ is eventually followed by an occurrence of ψ, the seventh pattern states that ψ_1 and ψ_2, in that order, precede the first occurrence of ϕ, the eighth pattern states that ψ_1 and ψ_2, in that order, follow every occurrence of ϕ, and the last pattern expresses that every occurrence of

Table 2. Property patterns for the global scope(ϕ is the place-holder for the absent event)

property	logic	pattern
absence	CTL	$\mathbf{AG}\neg\phi$
	LTL	$\Box\neg\phi$
universality	CTL	$\mathbf{AG}\phi$
	LTL	$\Box\phi$
existence	CTL	$\mathbf{AF}\phi$
	LTL	$\Diamond\phi$
bounded existence	CTL	$\neg\mathbf{EF}(\neg\phi \wedge \mathbf{EX}(\phi \wedge \mathbf{EF}(\neg\phi \wedge \mathbf{EX}(\phi \wedge \mathbf{EF}(\neg\phi \wedge \mathbf{EX}\phi)))))$
	LTL	$\neg\phi\,\mathcal{W}(\phi\,\mathcal{W}(\neg\phi\,\mathcal{W}(\phi\,\mathcal{W}\,\Box\neg\phi)))$
precedence	CTL	$\mathbf{A}[\neg\phi\mathbf{W}\psi]$
	LTL	$\neg\phi\,\mathcal{W}\,\psi$
response	CTL	$\mathbf{AG}(\phi \Rightarrow \mathbf{AF}\psi)$
	LTL	$\Box(\phi \Rightarrow \Diamond\psi)$
precedence chain	CTL	$\neg\mathbf{E}[\neg\phi\mathbf{U}(\psi_1 \wedge \neg\phi \wedge \mathbf{EXEF}\psi_2)]$
	LTL	$(\Diamond\phi) \Rightarrow (\neg\phi\,\mathcal{U}(\psi_1 \wedge \neg\phi \wedge o(\neg\phi\,\mathcal{U}\,\psi_2))$
response chain	CTL	$\mathbf{AG}(\phi \Rightarrow \mathbf{AF}(\psi_1 \wedge \mathbf{AXAF}\psi_2))$
	LTL	$\Box((\psi_1 \wedge \bigcirc\Diamond\psi_2) \Rightarrow \bigcirc\Diamond(\psi_2 \wedge \Diamond\phi))$
constrained chain	CTL	$\mathbf{AG}(\phi \Rightarrow \mathbf{AF}(\psi_1 \wedge \neg\psi_3 \wedge \mathbf{AXA}[\neg\psi_3\mathbf{U}\psi_2]))$
	LTL	$\Box(\phi \Rightarrow \Diamond(\psi_1 \wedge \neg\psi_3 \wedge \bigcirc(\neg\psi_3\,\mathcal{U}\,\psi_2)))$

ϕ is eventually followed by ψ_1 and ψ_2, in that order, without occurrence of ψ_3 between these occurrences of ψ_1 and ψ_2.

Although their expressiveness makes them good candidates to describe the behaviour of interacting systems, these temporal logics can be considered too low-level to be manipulated by the design engineers. Important efforts are being carried out to provide more concise temporal notations that can then be automatically mapped either to CTL or to LTL. In particular, the hardware design community has made good progress in this direction with the property specification language PSL [5].

Exercise 5. Identify instance of the property patterns presented in this section that can be used to specify expected properties of the systems modeled by Kripke structures $ABPsender$ and $ABPreceiver$.

3.4 Fairness Constraints

In a transition system such as a Kripke structure, some executions may be considered as unfair. For instance, in the case of the $ABPsender$, the path $s_0(s_1\,s_2)^\omega$ corresponds to a scenario where a sent message is never acknowledged, which would be caused by a defect in the environment of the modelled system. To verify that the system satisfies some of its properties, however, we may need to make some assumptions about the environment. Such assumptions are called *fairness constraints*.

Several ways of defining fairness constraints have been proposed. One of them is identifying a set of states that is visited infinitely often. For instance, in the case of our example, identifying the set of states $\{ s_0, s_3 \}$ as a fairness constraint can be used to restrict the behaviour of interest to the paths that pass through this set infinitely often, thus excluding from the model the paths that have $(s_1\, s_2)^\omega$ or $(s_4\, s_5)^\omega$ as a suffix.

Definition 5 (Fair Kripke structure, fairness constraint). *A* fair Kripke structure *over P is a quadruple $M = (S, T, I, L, F)$ where (S, T, I, L) is a Kripke structure and $F \subseteq 2^S$ is a set of sets of states. Each set of states $f \in F$ is called a* fairness constraint. *A path π is* fair *if, each $F_i \in F$ is visited infinitely often. Formally,*

$$\forall\, F_i : F \bullet \exists f : F_i \bullet \forall\, i \bullet \exists j \mid j > i \bullet f = \pi(j).$$

This type of fairness constraints is known as (generalized) Büchi acceptance conditions.

Exercise 6. Identify reasonable fairness constraints for *ABPreceiver*, or argue why no such constraints shall be imposed.

3.5 Quantitative Temporal Logic

CTL* and its derived temporal logics are based on a discrete notion of time. Each moment corresponds to a state of the system and the next moment corresponds to one of the possible successors of the state at the current moment. Execution is thus a sequence of states and the structure of time is that of the natural numbers.

As we have seen, the state quantification operators **F**, **G** and **U** are *qualitative*, in the sense that they make it possible to relate the occurrence of events, but are not *quantitative*, in the sense that they do not impose limits on the distance between such events. With the help of the **X** operators, it is possible to put bounds on the required relationships between events (see, for example, the bounded existence pattern in Table 2). However, the formulae are awkward and quickly become hard to express correctly.

To overcome this difficulty, it is possible to adopt some syntactic sugaring by adding operators annotated with temporal constraints. For instance, the logic RT-CTL [85] contains quantitative temporal operators, such as $\mathbf{AF}^{\leq c}$, stating that some event must occur within at most c steps, where c is some positive integer. As these new operators are syntactic sugar for combinations of the basic operators we have already seen, the traditional model-checking algorithms presented in Sections 4 and 5 apply. It is worth noting that specific algorithms, similar to the standard model-checking algorithms, have been proposed to handle a class of operators that return quantitative information about the model, such as the maximum delay between two events [44, 45].

Another possible model is that of dense, or continuous time, where the temporal structure is that of the (positive) real or rational numbers [8, 112]. The model of computation is that of *timed automata* and is more expressive than Kripke

structure, since, in addition to scalar variables, the state may also include *timers*. Timers are real (or rational)-valued variables that model the passing of time. All timers thus evolve at the same rate. Transitions may be triggered by a timer reaching a bound, and may also reset the value of timers to zero. Model-checking techniques for this class of systems require completely different approaches to that of Kripke structures, based on a data structure known as *difference bound matrices*. (Refer to [23] for a tutorial on those techniques and an introduction to their implementation in the tool UPPAAL [22].)

4 LTL Model Checking

The model-checking problem for LTL consists in, given a Kripke structure M over a finite set of atomic propositions AP, and an LTL formula ϕ over AP, decide if $M \models \phi$. The LTL model-checking approach is based on the following observations [249]:

1. First, an execution path in a Kripke structure is nothing but an infinite sequence of boolean assignments to the finite set of propositions AP. Now, consider the set $\Sigma_{AP} = \mathbb{P}\, AP$ of all such possible boolean assignments, then a path is an *infinite* word over this set.
2. Second, an LTL formula ϕ characterizes a set of infinite words (that is, paths) $\mathcal{L}(\phi)$ that satisfy it.
3. Given an LTL formula ϕ, it is possible to build an automata $\mathcal{A}(\phi)$ that accepts all the words in $\mathcal{L}(\phi)$. Such automata are called Büchi automata.
4. It is straightforward to build a Büchi automaton that accepts all the paths of a given Kripke structure.
5. If the composition of the Büchi automata for an LTL formula $\mathcal{A}(\phi)$ and for a Kripke structure M has no accepting path, then $M \models \neg\phi$.

To adopt an automata-based approach to linear temporal logic verification we need to know, first, how to compose two existing Büchi automata such that the resulting automaton accepts all infinite words accepted by both automata, and, second, how to check if a Büchi automaton accepts at least one infinite word.

In the remainder of this section, we will first present Büchi automata (Section 4.1. Then we show how to build a Büchi automaton that accepts all paths of a given Kripke structure (Section 4.2). We then explain how to build a Büchi automaton that accepts all paths that satisfy a given LTL formula (Section 4.3). We will then be ready to present the automata composition operator and the emptiness verification algorithm (Section 4.4). We conclude by presenting the widely popular LTL model-checker SPIN (Section 4.5).

4.1 Büchi Automata

This new form of automata was first introduced and defined in [36]. The definition is similar to that of finite automata (the type of machine that recognizes finite words), but the notion of acceptance has been modified to make it possible to recognize sets of infinite words (that is, ω-regular expressions).

Definition 6 (Büchi automaton). *A* Büchi automaton *is characterised by a tuple* $A = (\Sigma, S, T, I, F)$ *where,*

- Σ *is an alphabet;*
- S *is a set of states;*
- $T : S \times \Sigma \to \mathbb{P}\, S$ *is a nondeterministic transition function;*
- $I \subseteq S$ *is a set of initial states;*
- $F \subseteq S$ *is a set of accepting states.*

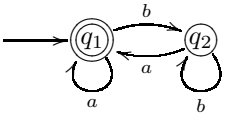


Fig. 6. A simple deterministic Büchi automaton

Example 4. Figure 6 [66] presents a simple Büchi automaton over the alphabet $\{ a, b \}$. The set of states is $\{ q_1, q_2 \}$, and the sets of initial and accepting states are both $\{ q_1 \}$. If we consider the definition of traditional finite-state automata, it recognizes the language denoted by the regular expression $\varepsilon + (a + b)^* a$, composed by all words ending with a and the empty word.

Definition 7 (Run, accepting run, acceptance). *A* run *of a* Büchi automaton A *over an infinite word* $w = a_1 a_2 \ldots$ *over* Σ *is a sequence* $s_0, s_1, \ldots$ *such that* $s_0 \in I$ *and* $\forall\, i \mid i \geq 1 \bullet s_i \in T(s_{i-1}, a_i)$. *An* accepting run *over* A *is a run* $s_0, s_1, \ldots$ *if there is some accepting state that repeats infinitely often, that is,*

$$\exists f : F \bullet \forall\, i \bullet \exists j \mid j > i \bullet f = s_j.$$

The word w *is said to be* accepted *by* A *if there is an accepting run of* A *for* w. *The language of* A, *denoted* $\mathcal{L}(A)$, *is the set of infinite words accepted by* A.

Example 5. The automaton of Figure 6 accepts, for example, the words a^ω (infinitely a), and $(ab)^\omega$ (alternating a and b infinitely, starting with a). It also accepts all words that contain an infinite number of occurrences of the symbol a, which is denoted by the ω-regular expression $(b^* a)^\omega$.

In a Büchi automaton, the transitions are defined by a function to a set of states. For a given pair (s, a), if $T(s, a)$ has a null cardinality, then there is no successor for state s and symbol a, and we say that the transition function is not total. If the cardinality is greater than one, then there are several possible successors, and the automaton is nondeterministic. In all these cases, the definition of acceptance applies.

Büchi automata recognize infinite words when a run goes through the set of accepting states; if there are more than one accepting state, we have no guarantee that an accepting run will visit each the accepting states. However, in general, to map a LTL formula to a Büchi automaton, we need to constrain the runs to go through several *different* states. For this purpose, we introduce the following definition.

Definition 8 (Generalized Büchi automaton). *A* generalized Büchi automaton *is a tuple* $A = (\Sigma, S, T, I, F)$ *where* Σ, S, T, I, *are as in a Büchi automaton, and* $F \subseteq \mathbb{P}\, S$ *is a set of sets of states. An* accepting run *over a generalized Büchi automaton* A *is a run* $s_0, s_1, \ldots$, *such that each* $F_i \in F$ *is visited infinitely often, that is:*

$$\forall\, F_i \in F \bullet \exists f \in F_i \bullet \forall\, i \bullet \exists j > i \bullet f = s_j.$$

This more general definition does not increase the class of languages that can be recognized. Indeed, given a generalized Büchi automaton (Σ, S, T, I, F), with $F = \{ F_1, \ldots F_n \}$, it is possible to construct a Büchi automaton that recognizes the same language, namely, $(\Sigma, S \times \{ 0, \ldots n \}, T', I \times \{ 0 \}, S \times \{ n \})$, where the transition relation T' is such that:

- $((s, x), a, (s', x + 1)) \in T'$ iff $(s, a, s') \in T$ and $s' \in F_{x+1}$;
- $((s, x), a, (s', x)) \in T'$ iff $(s, a, s') \in T$ and $s' \notin F_{x+1}$;
- $((s, n), a, (s', 0)) \in T'$ iff if $(s, a, s') \in T$.

The equivalent Büchi automaton is in a state (s, x) when the corresponding generalized Büchi automaton would be in state s after having visited the first x fairness conditions since the beginning of the execution, or since it has last visited all fairness conditions. The second component in the state of the Büchi automaton records the generalized fairness conditions that have already been visited. When the $(x + 1)$-th fairness condition is reached, then this index is incremented (first rule above), otherwise it retains its old value (second rule). Finally, when all fairness conditions have been traversed and $x = n$, then the index is set to 0 (last rule). The fairness condition of the equivalent Büchi automaton is the set of states where the second component is n. Thus this construction guarantees that all fairness conditions of the original generalized Büchi automaton are visited infinitely often.

Finally, we will consider a variation of generalized Büchi automata where each state carries a set of labels. This addition does not fundamentally change the purpose of Büchi automata, but will prove useful to provide a procedure that builds a Büchi automaton that accepts the paths that satisfy a LTL expression.

Definition 9 (Labeled generalized Büchi automaton). *A* labelled generalized Büchi automaton *is a triple* (A, D, L), *such that* $A = (\Sigma, S, T, I, F)$ *is a generalized Büchi automaton,* D *is a finite domain, and* $L : S \to \mathbb{P}\, D$ *is a mapping from states to sets of labels. A word* $x_0 x_1 x_2 \ldots \in D^\omega$ *is accepted if, and only if, there is an accepting run* $s_0 s_1 s_2$ *of* A *such that*

$$\forall\, i \mid i \geq 0 \bullet x_i \in L(s_i).$$

Specification with Büchi Automata. Büchi automata can be used to specify properties of the dynamic behaviour of systems. In this context, an execution run is an infinite word on the alphabet composed of the possible states of the system. We illustrate this point with two classes of properties, namely safety

and liveness. Figure 7 shows two automata that specify the set of execution paths in which p is always valid (safety), and in which p must always eventually happen (liveness). Note how the duality between the two properties is reflected in the automata. Indeed, some researchers advocate the use of automata-based specification languages over logic-based notations [6, 141].

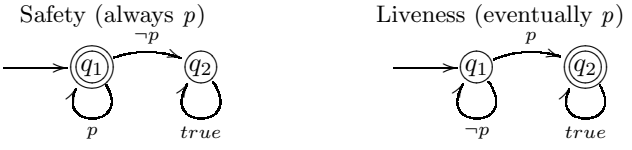


Fig. 7. Büchi automata for specifying temporal properties

Exercise 7. Use a Büchi automaton that recognizes execution paths where the condition p is recurring, that a condition p that occurs infinitely often (expressed in LTL as $\mathbf{GF}p$).

4.2 From Kripke Structures to Büchi Automata

The mapping from Kripke structures to a Büchi automata is straightforward. Let $M = (S, T, I, L)$ be a Kripke structure on AP. The Büchi automata that recognises the words that correspond to paths in M (and only those words) is $\mathcal{A}(M) = (\mathbb{P}\, AP, S \cup \{\, i \,\}, \mathcal{T}, \{\, i \,\}, S)$, where $i \notin S$ (i is a new state), $s \in \mathcal{T}(i, a)$ for all $s \in I$ and $a \in L(s)$, and $s' \in \mathcal{T}(s, a)$ if, and only if, $(s, s') \in T$ and $a \in L(s')$.

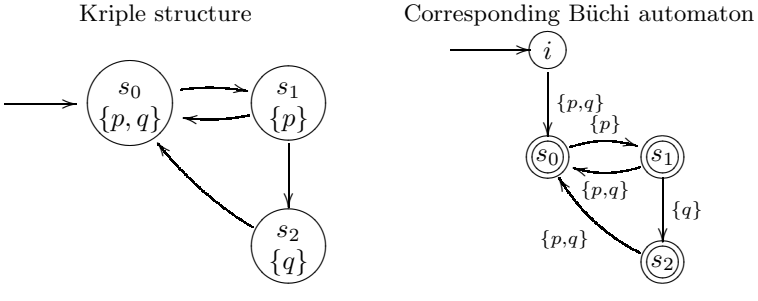


Fig. 8. The Büchi automaton of a Kripke structure

Example 6. An example of such correspondence is presented in Figure 8. Observe that the Büchi automaton has a corresponding state for each state of the Kripke structure as well as an extra initial state, labeled with i. There is a transition from the initial state of the Büchi automaton to the state representing the initial state of the Kripke structure. The other transitions of the Büchi

automaton correspond to transitions of the Kripke structure. The label of the Büchi automaton transitions are the same as that of the Kripke structure state associated with their target state. Observe that the transition function is not total.

4.3 From LTL to Büchi Automata

Recall that given an LTL formula φ, we want to build a Büchi automaton that recognizes all paths where φ is valid. The original paper on automata-based verification [249] provides an algorithm that solves this problem. Basically, it consists in constructing and composing two automata. The states of these automata are labelled with sets of sub-formulae of φ or their negation. The construction of the first automaton guarantees the semantics of the **X**operator (so-called local consistency), while the second automaton takes care of the **U**sub-formulae.

The drawback of this approach is that it always has the worst-case complexity. Considering that the complexity of the model-checking algorithm is linear in the size of the automaton (number of states), this approach is not practical, and several other alternatives and improvements have been proposed [99, 87, 244, 98].

We present here the algorithm of [99]. In that work, the generation of the Büchi automaton that accepts all paths that satisfy a given LTL formula φ is done by first putting φ into a convenient form φ', and then unfolding the state graph of the automaton from the initial state, labelling each created state with the sub-formulae of φ' that are valid at this point and those that will be established in the future. Once the state graph has been built, transitions are labelled and accepting states identified.

Normalization of LTL Formulae. We introduce the $\mathbf{V}$ operator satisfying:

$$\varphi \,\mathbf{V}\, \psi = \neg((\neg\varphi)\mathbf{U}(\neg\psi))$$

The LTL formula is first put into negation normal form: the boolean operators are translated to $\vee$ and $\wedge$, and all instances of the negation operator are pushed inwards to the front of the atomic propositions. The rewriting rules are as follows, where φ and ψ are LTL formulae:

$$\begin{array}{rr} \neg \mathit{true} \rightsquigarrow \mathit{false}, & \neg \mathit{false} \rightsquigarrow \mathit{true}, \\ \varphi \Rightarrow \psi \rightsquigarrow \neg\varphi \vee \psi, & \neg\mathbf{X}\varphi \rightsquigarrow \mathbf{X}\neg\varphi, \\ \mathbf{F}\varphi \rightsquigarrow \mathit{true}\mathbf{U}\varphi, & \mathbf{G}\varphi \rightsquigarrow \mathit{false}\,\mathbf{V}\,\varphi, \\ \neg(\varphi\mathbf{U}\psi) \rightsquigarrow (\neg\varphi)\,\mathbf{V}(\neg\psi), & \neg(\varphi\,\mathbf{V}\,\psi) \rightsquigarrow (\neg\varphi)\mathbf{U}(\neg\psi) \end{array}$$

Exercise 8. What is the normal form of the LTL formula $\mathbf{GF}p$?

Construction of the State Graph. We now present the (non-optimized) approach of [99] to build a labelled generalized Büchi automaton (LGBA) corresponding to an LTL formula φ in negation normal form. In this approach, the

state-transition graph is constructed incrementally. To perform this construction, the following information is associated to each graph node N: $N.id$ is a unique identifier; $N.todo$ is a set of LTL formulae that have to be established in N; $N.lbl$ is a set of LTL formulae that have been established for N; $N.in$ is a set of nodes that have a transition leading to N; and $N.next$ is a set of formulae that have to be established for all successors of N. By convention, the special symbol ι is a member of $N.in$ whenever N is an initial state of the LGBA. We also use the notation $[i, S_i, S_t, S_d, S_n]$ to denote the node N such that $N.id = i$, $N.in = S_i$, $N.todo = S_t$ and $N.next = S_n$.

The algorithm in Figure 9 (function *graph*) unfolds recursively the state-transition graph, starting from an initial node, say N_0, where formula φ shall be recognized (line 03). Thus we have:

$$N_0.todo = \{\, \varphi \,\} \wedge N_0.in = \{\, \iota \,\} \wedge N_0.lbl = \varnothing \wedge N_0.next = \varnothing.$$

We now consider function *unfold*. It takes as argument a node N to be explored and a set of nodes S that have already been fully explored.

If there is no more formulae that have to be established for N (established by the test $N.todo = \varnothing$, line 08), then the algorithm tests if a node N' with the same set of established formulae in the current state and in the successor states has already been built. If this is the case, then it is basically a clone of N, and the set of predecessors of N' is updated with that of N (lines 09–11). If there is no such clone node, then the algorithm proceeds by unfolding the state-transition graph from a new node that shall establish the formulae in $N.next$ and that shall contain N among its predecessors (line 13).

If the initial test (line 08) fails, the algorithm proceeds building the graph to establish one formula φ taken from $N.todo$. The rest of the algorithm depends on the nature of φ.

- If φ is a constant or a literal, the algorithm checks if it is compatible with the formulae in $N.lbl$, to avoid contradictions (lines 19–20).
- If $\varphi = \mathbf{X}\psi$, then ψ must be established in all successors of N, and therefore is added to $N.next$.
- If $\varphi = \psi_1 \wedge \psi_2$, then both ψ_1 and ψ_2 are added to the set of formulae that need to be established in N, and are added to $N.todo$.

In these first three cases, φ is added to $N.lbl$ and the algorithm proceeds recursively to establish the remainder of the formulae in $N.todo$ (lines 21–24).

If none of the three above cases apply, φ is a disjunction, either explicitly or implicitly (considering that $\psi_1 \mathbf{U} \psi_2 = \psi_2 \vee (\psi_1 \wedge \mathbf{X}(\psi_1 \mathbf{U} \psi_2)$, and that $\psi_1 \mathbf{V} \psi_2 = \psi_2 \wedge (\psi_1 \vee \mathbf{X}(\psi_1 \mathbf{V} \psi_2)) = (\psi_2 \wedge \psi_1) \vee (\psi_2 \wedge \mathbf{X}(\psi_1 \mathbf{V} \psi_2))$. The algorithm then creates a case-split in the graph, substituting N with two nodes N_1 and N_2, and each argument of the disjunct is added to $N_1.todo$ and $N_2.todo$.

Finally, we assume the existence of a function *nid* that yields a new state identifier each time it is invoked. Further, we define the functions arg_1 and arg_2 as follows:

```
01 function graph(φ: LTL): void
02 begin
03    unfold([nid(), { init }, ∅, { φ }, ∅], ∅)
04 end function graph
05
06 function unfold(N: node, S: set of node): void
07 begin
08    if N.todo = ∅ then
09       if ∃ N' : S • N.lbl = N'.lbl ∧ N.next = N'.next then
10          N'.in := N'.in ∪ N.in;
11          return
12       else
13          unfold([nid(), { N.id }, N.next, ∅, ∅], S ∪ { N })
14       end if
15    else
16       let φ ∈ N.todo in
17          N.todo := N.todo − { φ }
18          if φ is a literal or a constant then
19             if φ = true and not (¬φ) ∈ N.lbl then
20                N.lbl := N.lbl ∪ { φ };
21                unfold(N, S)
22             end if
23          else if φ = X ψ then
24             N.lbl := N.lbl ∪ { φ };
25             N.next := N.next ∪ { ψ };
26             unfold(N, S)
27          else if φ = ψ1 ∧ ψ2 then
28             N.todo := N.todo ∪ ({ ψ1, ψ2 } − N.lbl);
29             N.lbl := N.lbl ∪ { φ };
30             unfold(N, S)
31          else (* φ = ψ1 ∨ ψ2 ∨ φ = ψ1 U ψ2 ∨ φ = ψ1 V ψ2 *)
32             N1 := [nid(), N.in, N.todo ∪ (arg1(ψ) − N.lbl), N.lbl ∪ { φ }, N.next ];
33             N2 := [nid(), N.in, N.todo ∪ (arg2(ψ) − N.lbl), N.lbl ∪ { φ }, N.next ];
34             unfold(N1, unfold(N2, S))
35          end if
36       end let
37    end if
38 end function unfold
```

Fig. 9. Algorithm to construct a Büchi automaton recognising words satisfying an LTL formula [99]

$$
\begin{array}{ll}
arg_1(\psi_1 \mathbf{U} \psi_2) = \{ \psi_2 \}, & arg_2(\psi_1 \mathbf{U} \psi_2) = \{ \psi_1 \wedge \mathbf{X}(\psi_1 \mathbf{U} \psi_2) \}, \\
arg_1(\psi_1 \mathbf{V} \psi_2) = \{ \psi_2 \wedge \psi_1 \}, & arg_2(\psi_1 \mathbf{V} \psi_2) = \{ \psi_2 \wedge \mathbf{X}(\psi_1 \mathbf{V} \psi_2) \}, \\
arg_1(\psi_1 \vee \psi_2) = \{ \psi_1 \}, & arg_2(\psi_1 \vee \psi_2) = \{ \psi_2 \}.
\end{array}
$$

This algorithm halts and returns a set of expanded nodes from which a LGBA is built. First, the set of states is identified to the set of nodes. A state is initial

if $\iota \in N.in$. There is a transition from N to N' if $N \in N'.in$. The domain of the LGBA is $\mathbb{P}\,AP$. The label of a state N are the sets that are compatible with $N.lbl$. The accepting conditions are related to the sub-formulae of φ that have the form $\psi_1 \mathbf{U} \psi_2$. For each such sub-formula, there will be an acceptance set composed of the states N such that either $\psi_1 \mathbf{U} \psi_2 \notin N.lbl$ or $\psi_2 \in N.lbl$. This guarantees that in each accepting run of the LGBA, every occurrence of a state labelled with $\psi_1 \mathbf{U} \psi_2$ is followed by an occurrence of a state labelled with ψ_2.

Exercise 9. Apply the algorithm 9 to the LTL formula $\mathbf{GF}p$.

4.4 Operations on Büchi Automata

Composition of Automata. As explained earlier on, we need a definition for the composition of two Büchi automata, such that the resulting automaton accepts a word w if and only if it is accepted by both automata [57].

Definition 10 (Composition of Büchi automata). *Given two Büchi automata $A_1 = (\Sigma, S_1, T_1, I_1, F_1)$ and $A_2 = (\Sigma, S_2, T_2, I_2, F_2)$ on the same alphabet Σ, the composition of A_1 and A_2 is the Büchi automaton (Σ, S, T, I, F) such that:*

- $S = S_1 \times S_2 \times \{\, 0, 1, 2 \,\}$;
- $(s'_1, s'_2, k') \in T((s_1, s_2, k), a)$ *when* $s'_1 \in T_1(s_1, a)$ *and* $s'_2 \in T_2(s_2, a)$, *and*
  - *if* $k = 0$ *and* $s'_1 \in F_1$ *then* $k' = 1$;
  - *if* $k = 1$ *and* $s'_2 \in F_1$ *then* $k' = 2$;
  - *if* $k = 2$ *then* $k' = 0$;
  - *otherwise* $k' = k$.
- $I = I_1 \times I_2 \times \{\, 0 \,\}$;
- $F = S_1 \times S_2 \times \{\, 2 \,\}$.

Observe that the third state component in the resulting automaton, hereafter called the fairness counter, tracks the acceptance conditions with respect to the original automata. By convention and construction of the transition relation of the composition automaton, the fairness counter is 0 when fairness conditions of both original automata need to be met, 1 when only the fairness condition of the second automaton remains to be met, and 2 to indicate that both fairness conditions have been met. Since both conditions need to be met infinitely often, when the fairness status is 2, then it is reset to 0, and the fairness condition of the composition is 2.

Example 7. Figure 10 presents the result of the composition of two Büchi automata A_1 and A_2. A_1 recognizes words with an infinite number of a, while A_2 recognizes words with an infinite number of b. The resulting automata recognizes words with an infinite number of a and b.

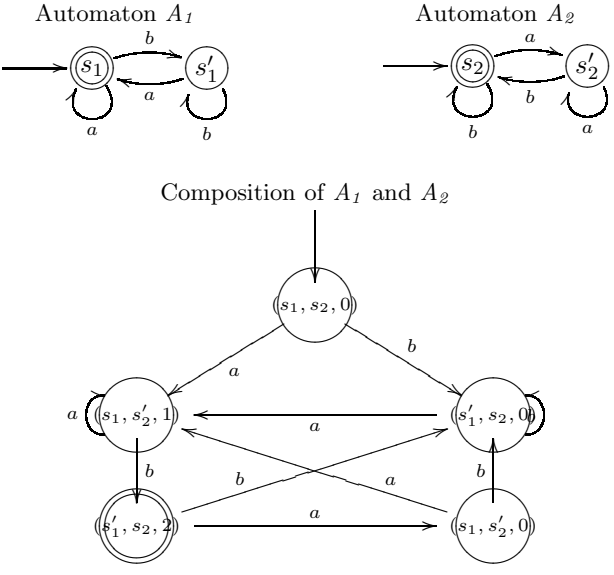


Fig. 10. Composition of two Büchi automata

Verifying Emptiness in Büchi Automata. All that we need now to perform LTL model-checking is to understand how to check that the set of accepting runs of a Büchi automaton is empty. If it is not empty, then we would like to obtain a counterexample, that is, an infinite sequence of transitions that goes through the set of accepting states infinitely often.

As the set of states of the automaton is finite, any accepting run must start from an initial state and contain a suffix that passes infinitely often through a finite number of states: these states form a strongly connected component of the transition graph. Furthermore, if the run is accepting, then this strongly connected component must pass through at least one accepting state infinitely often. In conclusion, we want to find a strongly connected component of the transition graph that is reachable from an initial state and that contains at least an accepting state of the Büchi automaton. If there is no such component, then the automaton accepts no run. Otherwise, it can be used to build a counterexample.

The classic algorithm to find a cycle in a graph is presented in[245]. The algorithm given in Figure 11, however, is more efficient in practice for the type of graphs that are produced in automata-based verification efforts [121], as it only requires an overhead of two bits per state, while the classical algorithm requires two numbers. Considering that automata arising in model checking contain billions of states, the difference of memory requirements is substantial.

The algorithm is composed of two nested depth-first traversals. We assume that there is a unique initial state ι. If this is not the case, a new initial state is created with transitions to the original initial states. We also assume that, for each state, flags f_o and f_i are associated to the outermost and innermost depth-first searches, respectively. Each depth-first search also maintains a stack

```
function find_cycle(M: Kripke structure): void
begin
  for all σ ∈ M.S do f_o(σ), f_i(σ) ← false, false end for;
  Σ_o := ∅;
  for all σ ∈ M.Q_0 do dfs_o(σ) end for;
  terminate(false)
end function dfs_o

function dfs_o(M: Kripke structure, σ: M.S): void
begin
  f_o(σ) := true;
  push(Σ_o, σ);
  for all σ' ∈ M.T(σ) do dfs_o(σ') end for;
  pop(Σ_o)
end function dfs_o

function dfs_i(M: Kripke structure, σ: M.S): void
begin
  f_i(σ) := true;
  for all σ' ∈ M.T(σ) do
    if σ' ∈ Σ_o then
      terminate(true)
    else if ¬f_i(σ')
      dfs_i(σ'); f_i(σ) := false
    end fi
  end for
end function dfs_i
```

Fig. 11. Nested-DFS algorithm to find one cycle in the state transition graph

of visited states, called Σ_o for the outermost search, and Σ_i for the innermost search. The outermost depth-first search finds all states reachable from the initial state. When a state σ is accessed, $f_o(\sigma)$ is set and σ is pushed on Σ_0. Note that Σ_o contains (one of) the shortest path(s) leading to σ. If σ is accepting, then the second, innermost, depth first search is started, looking for a cycle starting from σ. Note that it is enough to find a path from σ to one of the states in Σ_o. The innermost search similarly maintains flags f_i and stack Σ_i. If the innermost search finds a state σ' on Σ_o then it halts successfully. The counterexample is then composed of the prefix Σ_o from ι to σ and the cycle composed of the states in Σ_i plus the states of Σ_o from σ' up to σ.

4.5 The SPIN Model-Checker

SPIN [120] is a model checker for LTL that implements the approach described in this section with several optimizations that make it one of the most powerful and successful tools of its kind. Its first versions date back to the early eighties, and it has been the subject of many enhancements since then.

SPIN has a quite expressive input language, named PROMELA (a Process Meta Language), suitable to describe systems of asynchronous communicating processes. Specifications can be expressed in LTL and automatically verified. When a property is not satisfied, SPIN can provide a counterexample that can be displayed as a Message Sequence Chart and interactively simulated. The architecture of the system is presented in Figure 12.

For the less experienced user, the graphical front-end XSPIN is convenient to interact with the tools, as it provides default values for the many options they support. A high-level model of the system is described in PROMELA. The compilation process highlights possible syntax errors or generates the data structures for exploratory simulation runs. Once the designer is confident that the most trivial bugs have been removed, he can specify intended temporal properties in LTL. SPIN then generates optimized source code for the model checker tailored to the structure and properties being verified.

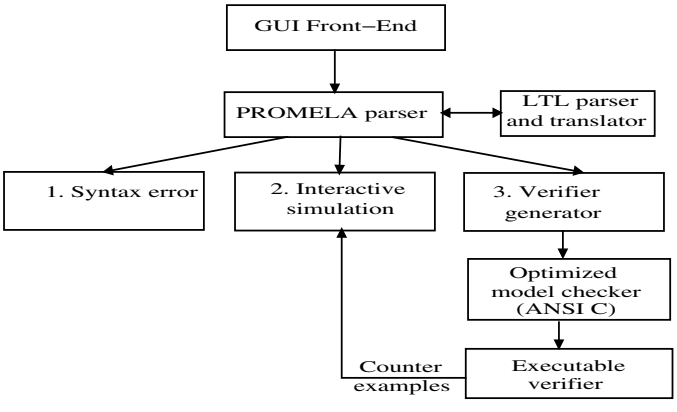


Fig. 12. Architecture of SPIN

The main focus of SPIN is on the interaction of processes. It provides several primitives to describe communications: rendezvous, shared variables, asynchronous message passing through channels, and combinations thereof. A program in PROMELA is composed of process templates, introduced with the keyword `proctype`, and at least one process instantiation. At run time, a process instantiation can dynamically spawn new instantiations of existing process templates. SPIN represents each process instance with one automaton. The whole system is composed of the interleaving of the execution of each process and is represented itself as a composition of the individual automata of each process. For model checking, the resulting automaton is composed with the automaton representing the negation of the property being verified, as explained in the previous sections.

To be able to tackle large examples, several improvements have been incorporated into SPIN. A partial-order reduction technique [211] is employed to identify

classes of states that can be considered equivalent with respect to the formula being verified. Indeed, the interleaving execution of processes often results in sequences of states that are equivalent from the viewpoint of the property being verified. Thus instead of exploring the whole state space, SPIN only builds a representative of each class of states.

Furthermore, SPIN adopts a so-called on-the-fly construction of the automaton modelling the system being verified. In the approach described in this chapter, both automata representing the PROMELA program and the (negation of the) LTL formula would need to be built before their composition is constructed and then checked for emptiness. The on-the-fly approach builds first the automaton representing the negation of the specification. The construction of the automaton representing the PROMELA program is guided by the specification automaton, following the rules of product automata construction. This technique avoids exploring large portions of the state space of the program automaton that have no counterpart in the specification automaton.

Two other techniques help limit the memory requirements of the verification process. First, state-compression [119] is a technique that makes it possible to use fewer bytes to represent each state of the automaton. The representation of the state is basically the concatenation of the representation of each state component: variables, process program counter, communication channel, and so on. State-compression uses a table for each such component to store its possible values. The representation of the state can then be the concatenation of the indices of the values of the corresponding state components in the tables. Important reductions in memory usage are obtained when the variables only evaluate to a subset of the possible range of values.

The second technique is bit-state hashing [118]. It sacrifices exhaustive coverage of the state space, at the expense of precision in the result. A bit vector is used to represent an estimation of the set of states. Each state is mapped to a position of the vector. When a state is visited, the corresponding bit is set. The state-space exploration thus checks if a state has been visited by checking if the corresponding vector position has already been set. A hash function defines the mapping from states to vector indices. Due to possible collisions, the state-space exploration is incomplete in general. In order to improve the precision of the technique, SPIN actually maintains not one, but two bit vectors, each being addressed by independent hash functions. In addition to the bit vectors, a depth-first traversal only has to store the stack of currently visited states. With carefully designed hash functions, this technique achieves very high coverage rates at a fraction of the memory necessary to represent exhaustively the state space.

Exercise 10. Access the SPIN site at `http://www.spinroot.com`. Download the tool and apply it to the provided examples.

5 CTL Model-Checking

Given a Kripke structure $M = (S, T, I, L)$ over a set of propositions P and a CTL formula f, the algorithm given in [63] works by associating to each state a

label. The label of a state is the set of the sub-formulae of f that are true in that state. The top-level algorithm recurses over the structure of the formula f and repeatedly labels the states of M with the sub-formulae of f, starting with the sub-formulae of length 1 (atomic propositions in P) and finishing with f itself. Figure 13 sketches this algorithm. Once this labelling phase has completed, the outcome of the algorithm is obtained by checking that f is in the label of each initial state of M.

The function *Emc* first calls the graph labeling function with the given formula and then checks that all initial states are labelled. The labelling function *Label* recurses down the structure of the formula: the function *args* returns the set of sub-formulae of a given CTL formula f. The terminal case is when the formula is an atomic function. If the formula is a boolean function, each state is labelled according to the previous labelling of the function arguments. Finally, to deal with temporal operators, special-purpose labelling functions are invoked.

```
function Emc(M: Kripke structure, f: CTL): boolean
begin
   Label(M, f);
   for all s ∈ I do
      if f ∉ L(s) then return false end if
   end for;
   return true
end function Emc

function Label(M: Kripke structure, f: CTL): void
begin
   if f ∈ P then return end if;
   for all f_i ∈ args(f) do Label(M, f_i) end for;
   case f of
      when ¬f_1:
         for all s ∈ S do
            if f_1 ∉ L(s) then L(s) := L(s) ∪ { f } end if
         end for
      when f_1 ∧ f_2:
         for all s ∈ S do
            if f_1 ∈ L(s) ∧ f_2 ∈ L(s) then L(s) := L(s) ∪ { f } end if
         end for
      when EX f_1: LabelEX(M, f_1)
      when EG f_1: LabelEG(M, f_1)
      when E[f_1 U f_2]: LabelEU(M, f_1, f_2)
   end case
end function Label
```

Fig. 13. Labeling algorithm for CTL formulae

The function *LabelEX* deals with $\mathbf{EX}f$ formulae and is given in Figure 14. For each transition $t = (s_1, s_2)$, if s_2 is labelled with f, s_1 has a successor where f is valid, and formula $\mathbf{EX}f$ is added to the label of the state s_1.

```
function LabelEX(M: Kripke structure, f: CTL): void
begin
  for all t = (s1, s2) ∈ T do
    if f ∈ L(s2) then L(s1) := L(s1) ∪ { EXf } end if
  end for
end function LabelEX
```

Fig. 14. Labeling algorithm for $\mathbf{EX}f$ formulae

Function *LabelEU* (see Figure 15) handles $\mathbf{E}[f\,\mathbf{U}\,g]$ formulae. They are valid in a state if and only if there is a finite path starting in this state, where f is always valid, except in the last state, where g is valid. First, each state already labelled with g is also labelled with $\mathbf{E}[f\,\mathbf{U}\,g]$. Then, the function *LabelEUaux* is invoked and backtracks along the transitions while f appears in the labels of the states. Each state found along such paths is labelled with formula $\mathbf{E}[f\,\mathbf{U}\,g]$. To avoid infinite loops, this backtracking also stops as soon as it meets a state already labelled with $\mathbf{E}[f\,\mathbf{U}\,g]$.

```
function LabelEU(M: Kripke structure, f: CTL, g: CTL): void
begin
  for al s ∈ S do
    if g ∈ L(s) then
      L(s) := { E[f U g] } ∪ L(s);
      for all t = (s', s) ∈ T do LabelEUaux(M, f, g, s') end do
    end if
  end for
end function LabelEU

function LabelEUaux(M: Kripke structure, f: CTL, g: CTL, s: state): void
begin
  if f ∈ L(s) ∧ E[f U g] ∉ L(s) then
    L(s) := L(s) ∪ { E[f U g] };
    for all (s', s) ∈ T do LabelEUaux(M, f, s') end do
  end if
end function LabelEUaux
```

Fig. 15. Labeling algorithm for $\mathbf{E}[f\,\mathbf{U}\,g]$ formulae

Finally, function *LabelEG* handles $\mathbf{EG}f$ formulae (see Figure 16). They are valid in a state s if, and only if, there is an infinite path, starting at s and where f holds in each state. To detect such situations, it is necessary to find cycles in the transition graph along which f is always valid. This is the role of the auxiliary function *LabelEGaux*; it has an additional parameter s, which is the state currently visited, and returns a boolean to indicate if $\mathbf{EG}f$ is valid in s. Additionally, two flags are associated with each state: *checked*(s) and *mark*(s). The first indicates if the algorithm has already computed if $\mathbf{EG}f$ is valid or not

in the state; $mark(s)$ is true if the algorithm has not yet checked if $\mathbf{EG}f$ holds in the state s, and if s starts a finite path along which f always hold. The function *LabelEGaux* performs a depth-first search along the transition graph as long as:

1. It does not reach a state s that has already been checked. If it does, it stops backtracking and returns a boolean that indicates whether $\mathbf{EG}f$ is valid in s or not.
2. It does not reach a state that is marked. If it does, then it has found a cycle where f is always valid. In this case it returns *true*.
3. It does not reach a state s where f does not hold. If it does, then the value returned is *false*. (This is implicit in this algorithm.)
4. Otherwise f is valid in s and $\mathbf{EG}f$ is potentially valid in s. The algorithm then calls itself recursively and checks if $\mathbf{EG}f$ is valid in one of the successors of f. As soon as one such successor is found, then the formula $\mathbf{EG}f$ is added to the label set of s and the algorithm returns immediately *true*. If no such successor is found, then formula $\mathbf{EG}f$ is not added to the label set and the algorithm returns *false*.

```
function LabelEG(M: Kripke structure, f: CTL): void
begin
   for all s ∈ S do checked(s) := false, mark(s) := false end for;
   for all s ∈ S do LabelEGaux(M, f, s) end for
end function LabelEG

function LabelEGaux(M: Kripke structure, f: CTL, s: state): boolean is
begin
   if ¬checked(s) then
      if mark(s) then return true end if;
      if f ∈ L(s) then
         mark(s) := true;
         for all (s, s') ∈ T do
            if LabelEGaux(M, s', f) then
               L(s) := L(s) ∪ { EGf }; checked(s) := true; return true
            end if;
         end for;
         mark(s) := false;
      end if;
      checked(s) := true;
   end if;
   return (EGf ∈ L(s))
end function LabelEGaux
```

Fig. 16. Labeling algorithm for $\mathbf{EG}f$ formulae

Example 8 (Verification of ABPsender). To illustrate the model-checking algorithm, we apply it to the verification of the *ABPsender* (see Figure 1). More specifically, we check that $\mathbf{EG}(\neg s \wedge \neg w)$ is a property of *ABPsender*; this is verified by the function call $Emc(ABPsender, \mathbf{EG}(\neg s \wedge \neg w))$.

The first step of the algorithm of *Emc* (se Figure 13) consists in labelling recursively the structure with the formulae and its sub-formulae. This is done by invoking the function *Label* with parameters $ABPsender$ and $\mathbf{EG}(\neg s \wedge \neg w)$). The function *Label* first labels the states of the graph with each one of the sub-formulae of the specification that are valid in those states. Since these sub-formulae are all boolean, the application of *Label* is trivial and yields the following state labelling.

$$
\begin{aligned}
L(s_0) &= \{\, g, \neg s, \neg w, \neg s \wedge \neg w \,\} \\
L(s_1) &= \{\, w, \neg s \,\} \\
L(s_2) &= \{\, s, \neg w \,\} \\
L(s_3) &= \{\, sg, b, \neg s, \neg w, \neg s \wedge \neg w \,\} \\
L(s_4) &= \{\, w, b, \neg s \,\} \\
L(s_5) &= \{\, s, b, \neg w \,\}
\end{aligned}
$$

Next, function *Label* invokes $LabelEG(ABPsender, \neg s \wedge \neg w)$ (see Figure 16). The flags *checked* and *mark* of each state are initialized to *false*. Then the function *LabelEGaux* is invoked on each state. We suppose that the first state to be inspected is s_0 and trace the call $LabelEGaux(ABPsender, \neg g \wedge \neg w, s_0)$. Since s_0 is not yet checked and $\neg g \wedge \neg w$ belongs to $L(s_0)$, then the *mark* flag of s_0 is changed to *true*, and for each transition leaving s_0, the function *LabelEGaux* is called on the destination. If we suppose that the transition (s_0, s_0) is chosen first, then *LabelEGaux* is invoked again on state s_0, tests the flag *mark*, which is now set, and returns *true*. The execution flow continues from the first invocation of *LabelEGaux* and adds $\mathbf{EG}\neg s \wedge \neg w$ to the set $L(s_0)$, sets true the *checked* flag, and returns true. This operation is repeated for each state of the structure. When *LabelEG* returns, the state labelling of $ABPsender$ is:

$$
\begin{aligned}
L(s_0) &= \{\, g, \neg s, \neg w, \neg s \wedge \neg w, \mathbf{EG}(\neg s \wedge \neg w) \,\} \\
L(s_1) &= \{\, w, \neg s \,\} \\
L(s_2) &= \{\, s, \neg w \,\} \\
L(s_3) &= \{\, g, b, \neg s, \neg w, \neg s \wedge \neg w, \mathbf{EG}(\neg s \wedge \neg w) \,\} \\
L(s_4) &= \{\, w, b, \neg s \,\} \\
L(s_5) &= \{\, s, b, \neg w \,\}
\end{aligned}
$$

The second part of *Emc* executes then and checks that for each initial state of $ABPsender$ the formula belongs to the label set, which is the case. Therefore, we conclude that $ABPsender \models \mathbf{EG}(\neg s \wedge \neg w)$.

To illustrate the original model-checking algorithm, a fully-functional implementation was demonstrated with a version of the complete alternating bit protocol that had a total of 251 states, with running times taking about 10 seconds for each formula to be verified [63]. After further optimizations, a parallel version of the model-checking algorithm, implemented on a vector architecture, was able to verify a Kripke structure with 131,072 states and 67,108,864 transitions, its

specification being a CTL formula with 113 sub-formulae. The time reported for this experiment was 225 seconds.

Despite these somehow impressive sounding results, in practice, the model checking presented above is not efficient enough to deal with industrial designs. In concurrent systems, the size of the state space grows exponentially with the number of components. For instance, the model of a sequential circuit with n flip-flops is a Kripke structure with potentially 2^n states: for $n = 32$, the order of magnitude of the number of potential states is 10^9. This phenomenon is known as the *state space explosion*, and makes it practically impossible to represent exhaustively the set of states and the set of transitions of most systems. A significant breakthrough was achieved to address this problem by introducing the idea of combining breadth-first traversal algorithms with a symbolic representation of the set of states and transitions. The next section presents the main ideas underlying symbolic model-checking of Kripke structures for the logic CTL.

Exercise 11. Apply the algorithms presented in this section to model check that $ABPsender \models \mathbf{AGEF}g$.

6 Symbolic Model-Checking

One possible way to avoid (or, at least, delay) the state space explosion is to represent sets of states and transitions by their characteristic function rather than by enumeration. It is the purpose of Section 6.1 to explain how propositional logic may be used as a language to define and manipulate the characteristic functions. In Section 6.2, we present binary decision diagrams (BDDs), an efficient graph-based implementation of propositional logic. Finally, Section 6.3 details the symbolic version of the model-checking algorithms presented in the previous section.

6.1 Kripke Structures and Propositional Logic

Representing States and Transitions. Let $M = (S, T, I, L)$ be a Kripke structure over $P = \{ v_1, \ldots, v_n \}$. The characteristic function of a state is a boolean function over the set of propositions P that provides a unique code for that state. The characteristic function is a conjunction of atoms. Each atom is either a variable v_i, in case v_i appears in the label of s, or the negation of v_i, otherwise. Formally, let $\mathbf{v}$ denote $(v_1, \ldots, v_n)$. The characteristic function of a state $s \in S$, denoted $[s]$, is defined as:

$$[s](\mathbf{v}) = \left(\left(\bigwedge_{v_i \in L(s)} v_i \right) \wedge \left(\bigwedge_{v_i \notin L(s)} \neg v_i \right) \right)$$

The definition of the characteristic function is extended to sets of states. The characteristic function of a set of states can also be viewed as a coding for this set. It is the conjunction of the characteristic functions of the states in the set. Formally, it can be defined as:

$$[\varnothing](\mathbf{v}) = \mathit{false}$$
$$[\{\, x \,\} \cup X](\mathbf{v}) = [x](\mathbf{v}) \vee [X](\mathbf{v})$$

The characteristic function of a transition $t = (s_1, s_2) \in T$ is constructed by conjunction of the characteristic functions of $[s_1]$ and $[s_2]$, where each atomic proposition has been renamed to a fresh proposition. Formally, let $P' = \{\, v'_1, \ldots v'_n \,\}$ be a set of fresh boolean propositions. The characteristic function of the transition i, denoted $[t]$, is defined as:

$$[t](\mathbf{v}, \mathbf{v}') = [s_1](\mathbf{v}) \wedge [s_2](\mathbf{v}')$$

This definition can be extended to represent sets of transitions as for sets of states, operating a conjunction on the characteristic functions of the individual transitions.

Example 9 (Characteristic function). In the Kripke structure *ABPsender*, the characteristic functions of the initial state s_0, of the transition (s_0, s_1) and of the initial states I are, respectively:

$$\begin{aligned}
[s_0] &= g \wedge \neg s \wedge \neg w \wedge \neg b \\
[(s_0, s_1)] &= (g \wedge \neg s \wedge \neg w \wedge \neg b) \wedge (\neg g' \wedge s' \wedge \neg w' \wedge \neg b') \\
[I] &= (g \wedge \neg s \wedge \neg w \wedge \neg b) \vee (g \wedge \neg s \wedge \neg w \wedge b) \\
&= (g \wedge \neg s \wedge \neg w) \\
[T] &= (b \underline{\vee} b') \wedge \neg g \wedge \neg s \wedge w \wedge g' \wedge \neg s' \wedge w' \\
&\quad \vee (b \leftrightarrow b') \wedge (g \wedge \neg s \wedge \neg w \wedge \neg w' \wedge (g' \underline{\vee} s')) \\
&\quad \vee (\neg g \wedge s \wedge \neg w \wedge \neg g' \wedge \neg s' \wedge w') \\
&\quad \vee (\neg s \wedge \neg g' \wedge s' \wedge \neg w' \wedge (g \underline{\vee} w))
\end{aligned}$$

Note that it is not the case that any conjunction of literals is the coding of a state. For instance $g \wedge s \wedge w \wedge b$ does not represent any state of the *ABPsender* Kripke structure. Such codings are said to be *unreachable*, as no sequence of transitions from an initial state can lead to such state. Codings corresponding to states are said to be *reachable*.

To simplify the notation, in the rest of this chapter, we will identify $[X]$ with X, where the definitions presented will be used to present the algorithms known as symbolic model checking. Also, as a slight abuse of language, codings of states and transitions are simply called states and transitions.

Exercise 12. What are the characteristic functions of the initial states and of the transition relation of the Kripke structure *ABPreceiver*?

State Space Traversal. Now that we have seen how propositional logic can be used to model sets of states and transitions, we will see how the traversal of the state space can be expressed using classic logic and fixpoint operators.

Let $M = (S, T, I, L)$ be a Kripke structure over P. The image of a set of states $X \subseteq S$ is the set of states that can be reached in one transition from X:

$$\{\, s \in S \mid \exists\, s' \in X \bullet (s', s) \in T \,\}$$

The characteristic function of the image of X, denoted $Forward(M, X)$, is:

$$Forward(M, X)(\mathbf{v}')=\exists\, \mathbf{v} \bullet X(\mathbf{v}) \wedge T(\mathbf{v}, \mathbf{v}') \tag{2}$$

Conversely, the inverse image of a set of states $X \subseteq S$ is the set of states from which X can be reached in one transition:

$$\{\, s \in S \mid \exists\, s' \in X \bullet (s, s') \in T \,\}$$

The characteristic function of the inverse image of a set of states X, denoted $Backward(M, X)$, is:

$$Backward(M, X)(\mathbf{v})=\exists\, \mathbf{v}', X(\mathbf{v}') \wedge T(\mathbf{v}, \mathbf{v}') \tag{3}$$

We now consider the case of reachable and unreachable codings. By definition, as the transition function characterizes all the transitions of the Kripke structure, the image of a reachable state is reachable. Henceforth, the inverse image of an unreachable state may not be reachable and is therefore necessarily unreachable.

6.2 Binary Decision Diagrams

Binary decision diagrams (BDDs, for short) are heuristically efficient data structures to represent formulae of the propositional logic. BDDs can be viewed as multi-rooted binary directed acyclic graphs. Each node n represents a different boolean function φ_n. Let φ be such a function, and p a proposition on which φ depends, $\varphi\langle p \leftarrow V\rangle$ denotes the function equal to φ where p has been replaced with the boolean constant V.

There are two kinds of BDD nodes: leaf nodes and non-leaf nodes. Leaf nodes represent constant boolean functions *true* and *false*. Non-leaf nodes represent non-constant boolean functions. A non-leaf BDD node n representing a function φ_n is labelled with a proposition p_n and has two edges, one that points to the formula $\varphi_n\langle p_n \leftarrow true\rangle$, and the other pointing to the formula $\varphi_n\langle p_n \leftarrow false\rangle$. Each node thus represents a decision, or a *test* on the associated proposition. Note that if φ_n did not depend on p_n, both edges would point to the same BDD node and the test, as well as the corresponding node, would be useless. To avoid wasting space with useless nodes, the BDD construction algorithm does not build them. Once the last of the argument propositions of a function has been tested, the edge must point to a node representing a constant boolean function, that is to a leaf. In a ML-like notation, the data type to represent a BDD can be defined as:

```
type bdd = Leaf of boolean |
           NonLeaf of (proposition * bdd * bdd);;
```

where `boolean` and `proposition` would be the types for booleans and propositions.

For instance, consider $P = \{a, b\}$ and a function $\varphi = a \wedge b$. As φ depends on both a and b, its BDD representation can either start with a test on a or a test on b. Suppose the test is on a, and let b_1 represent the corresponding BDD. As $\varphi\langle a \leftarrow \mathit{true}\rangle = b$ and $\varphi\langle a \leftarrow \mathit{false}\rangle = \mathit{false}$, the outgoing edges of this first test node point, respectively, to a second BDD b_2 that represents the function b and a third BDD b_3 that represents the constant function *false*. Constant functions are represented by leaf nodes, so b_3 is a leaf. Now let us consider BDD b_2. It represents the function b. This function only depends on proposition b, so the unique possible test is on proposition b: $b\langle b \leftarrow \mathit{true}\rangle$ and $b\langle b \leftarrow \mathit{false}\rangle$ are, respectively, the functions *true* and *false*. We have already seen that function *false* is represented by the leaf node b_3. So, one of the outgoing edges of b_2 points to b_3. The other edge points to a leaf node b_4 representing *true*. Figure 17 depicts the corresponding BDD (plain and dashed arrows indicate respectively assignment to *true* and *false*). Using our ML-like language, the nodes would thus be:

```
b3 = Leaf(false);;
b4 = Leaf(true);;
b2 = NonLeaf(b, b4, b3);;
b1 = NonLeaf(a, b2, b3);;
```

Observe that, in this example, we are initially faced with the choice of a proposition to test against (a or b?). Had we chosen to start testing against b, the resulting BDD construction would have been different. So, in order to facilitate the operations on BDDs, the BDD construction algorithm is constrained to follow a previously established *total order* on the propositions.

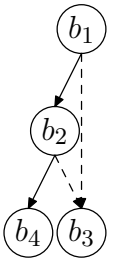


Fig. 17. Example BDD for function $a \wedge b$. The decision variable of b_1 is a, that of b_2 is b, b_3 is the leaf for *false* and b_4 is the leaf for *true*.

Let P be a totally ordered finite set of boolean propositions. Let f be a boolean formula over P, $bdd(f)$ is the BDD representing f, and $\mid bdd(f) \mid$ is the size of this BDD. The work presented in [34] showed that BDDs are a *canonical representation*, in the sense that two equivalent formulae are represented with the same BDD:

$$f \Leftrightarrow g \text{ iff } bdd(f) = bdd(g)$$

Moreover, most boolean operations can be performed efficiently with BDDs.

- $bdd(\neg f)$ is computed in constant time $O(1)$;
- $bdd(f \vee g)$ is realised in $O(\mid bdd(f) \mid . \mid bdd(g) \mid)$;
- $bdd(\exists\, x \bullet f)$ is performed in $O(\mid bdd(f) \mid^2)$.

In this chapter, we will use usual boolean operators to denote the corresponding operation on BDDs; for instance $bdd(f) \vee bdd(g) = bdd(f \vee g)$. We explain the basic principles of the BDD representation using an example.

Figure 18 presents the BDD of the characteristic function for the reachable states of Kripke structure $ABPsender$, with variable ordering $g < s < w < b$. Dotted edges indicate that the formula on the target node is negated. Therefore, the same BDD node is used to represent both a formula and its negation; it is interpreted differently according the type of edge that is pointing to it. (The BDD in Figure 18 represents $\neg g \vee s \vee w$ as well.) In practice, this "trick" can be performed using pointers to nodes to represent a BDD and pointer tagging to indicate negation. With variable ordering $g < g' < s < s' < w < w' < b < b'$, the BDD for the transition relation has 22 nodes.

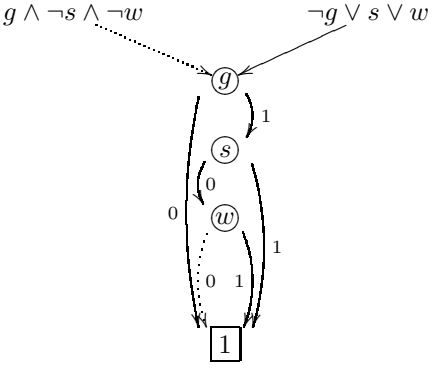


Fig. 18. BDD for $g \wedge \neg s \wedge \neg w$

[34] showed that some functions have an exponential BDD representation for any test order, and that finding the optimum variable ordering is NP-hard. In practice, however, heuristic methods generally achieve a good variable ordering, when such ordering exists. Moreover it is possible, although costly, to modify the test order. This investment to improve the BDD size is realized in situations were further boolean operations are going to be realized and pay it off.

In a Kripke structure, states, transitions and sets thereof can be characterized with propositional logic formulae. These formulae can be represented and manipulated via their BDD representation. BDDs proved to be an efficient data structure to perform computations on large Kripke structures.

In the remainder of this chapter, we will use the following operations on BDDs:

- *BddFalse* and *BddTrue* return the BDDs for the boolean constants;
- *BddAtom* takes a boolean proposition p as parameter and returns the BDD that represents p;
- *BddNot* takes as parameter the BDD of a boolean formula f and returns the BDD of formula $\neg f$;
- *BddAnd* (respectively, *BddOr*) take as parameters the BDDs of two boolean formulae f and g and returns the BDD of $f \wedge g$ (respectively, $f \vee g$);
- *BddImplies* is a predicate that takes as parameters the BDDs of two boolean formulae f and g and checks wether f is a logical implication of g.

These operations are used in the algorithms presented in the sequel.

Exercise 13. Consider the formula $(p_1 \Leftrightarrow q_1) \wedge (p_2 \Leftarrow q_2)$. Depict the corresponding BDD with variable ordering p_1, q_1, p_2, q_2. Repeat the exercise with variable ordering p_1, p_2, q_1, q_2. Discuss the effect of variable ordering on the BDD representation.

6.3 Algorithms

As explained in Chapter 1, a lattice is a set with a partial order on the elements of this set, a least element $\bot$, and a greatest element $\top$. Let P be a non-empty finite set of atomic propositions. Let $M = (S, T, I, L)$ be a finite Kripke structure over P. We consider the lattice $(\mathbb{P}\, S, \subseteq)$ of subsets of S with set inclusion as the ordering. The empty set $\varnothing$ and S are, respectively, the least and the greatest elements of this lattice. Since a subset of S can be identified with its characteristic function, this lattice can also be interpreted as the lattice of characteristic functions, with boolean implication as ordering, the characteristic function of $\varnothing$ as the least element, and the characteristic function of S as the greatest element.

With this view, we regard a function $\tau : \mathbb{P}\, S \to \mathbb{P}\, S$ as a predicate transformer. As in Chapter 1, τ is monotonic if, and only if, $P \subseteq Q$ implies $\tau(P) \subseteq \tau(Q)$. Also, τ is $\cup$-*continuous* (respectively, $\cap$-*continuous*) when $P_1 \subseteq P_2 \subseteq \ldots$ (respectively, $P_1 \supseteq P2 \supseteq \ldots$) implies that $\tau(\cup_i P_i) = \cup_i \tau(P_i)$ (respectively, $\tau(\cap_i P_i) = \cap_i \tau(P_i)$). If τ is monotonic, then τ has a least fixed point and a greatest fixed point [246].

Moreover, since the lattice is finite and τ is monotonic, it is also $\cup$-continuous and $\cap$-continuous, and least fixed points and greatest fixed points can be characterized as follows.

$$\mu\, Z \bullet \tau(Z) = \cap \{\, Z \mid \tau(Z) = Z \,\} = \cup_i \tau^i(\mathit{false}) \tag{4}$$

$$\nu Z \bullet \tau(Z) = \cup \{\, Z \mid \tau(Z) = Z \,\} = \cap_i \tau^i(\mathit{true}) \tag{5}$$

In the following, we will use these definitions to characterize the set of states satisfying a CTL formula with fixed point expressions.

Fixed Point Characterization of CTL Operators. CTL symbolic model-checking uses the BDD representations of the characteristic functions of sets of states and transitions. The algorithm is based on the fixed point characterization of the different temporal operators of CTL defined in [62]:

$$\mathbf{EG}f = \nu Z \bullet f \wedge \mathbf{EX}Z \tag{6}$$

$$\mathbf{E}[f\,\mathbf{U}\,g] = \mu Z \bullet g \vee (f \wedge \mathbf{EX}Z) \tag{7}$$

It is easy to see and show that, the lattice formed by the set of states and set inclusion is finite, and that, if $Z_1 \subseteq Z_2$, then $\mathbf{EX}Z_1 \subseteq \mathbf{EX}Z_2$. Thus, the predicate transformer $\mathbf{EX}Z$ is monotonic and so are the predicate transformers $f \wedge \mathbf{EX}Z$ and $g \vee (f \wedge \mathbf{EX}Z)$. Therefore, the fixed point expressions on the right-hand side of Equations 6 and 7 are well-defined.

To see that Equation 6 is indeed correct, assume that a state $s \models \mathbf{EG}f$. Then, applying the definition of the semantics of the operator **EG** (see Section 3.2), it is easy to show that $s \models f$ and that $s \models \mathbf{EXEG}f$, thus $s \in \nu Z \bullet f \wedge \mathbf{EX}Z$. Conversely, assume that $s \nu Z \bullet f \wedge \mathbf{EX}Z$, then $s \models f$ and $s \models \mathbf{EXEG}f$. Hence $s \models \mathbf{EG}f$. The same type of argument applies to justify Equation 7.

The symbolic model-checking algorithm, named *Smc*, is shown in Figure 19; it takes as arguments a Kripke structure M and a CTL formula f. It uses the auxiliary routine *SmcAux*, which returns the characteristic function of the states of M that satisfy f, and checks if f is valid in each initial state of M. The function *SmcAux* itself relies on auxiliary functions *SmcEX* (see Figure 20), *SmcEU* (see Figure 21) and *SmcEG* (see Figure 22). All of them are used to recurse over the syntactical structure of CTL formulae, and have as arguments a Kripke structure M and a CTL formula f, and as result the BDD of the characteristic function of the set of M states where f is valid.

```
function Smc(M: Kripke structure, f: CTL): boolean
begin
   return BddImplies(M.I, SmcAux(M, f))
end function Smc
function SmcAux(M: Kripke structure, f: CTL): BDD
begin
   case f of
      when f is a boolean proposition: return BddAtom(f)
      when f = ¬f1: return BddNot(SmcAux(M, f1))
      when f = f1 ∧ f2: return BddAnd(SmcAux(M, f1), SmcAux(M, f2))
      when f = EXf1: return SmcEX(M, f1)
      when f = E[f1 U f2]: return SmcEU(M, f1, f2)
      when f = EGf1: return SmcEG(M, f1)
   end case
end function SmcAux
```

Fig. 19. Algorithm for symbolic model-checking CTL formulae

SmcEX(M, f) computes the states of M where $\mathbf{EX}f$ is valid (Figure 20). Recall that a state s satisfies $\mathbf{EX}f$ if, and only if, a successor s' of s satisfies f.

So any predecessor of a state satisfying f satisfies $\mathbf{EX}f$, and any state satisfying $\mathbf{EX}f$ is a predecessor of some state satisfying f. Given the set of states S_f satisfying f, the set of states $S_{\mathbf{EX}f}$ satisfying $\mathbf{EX}f$ is the inverse image of S_f by the transition relation of the Kripke structure.

Thus the algorithm first computes F, the BDD for the characteristic function of the set states of M where f is valid, and returns the inverse image of F. The inverse image is computed with *Backward*, an implementation of the function defined by Equation 3.

```
function SmcEX(M: Kripke structure, f: CTL): BDD
variables
  F: BDD
begin
  F :=SmcAux(M, f):
  return Backward(M, F)
end function SmcEX
```

Fig. 20. Algorithm for $\mathbf{EX}f$ formulae

In a similar way, $SmcEU(M, f, g)$ computes the states of M where $\mathbf{E}[f\,\mathbf{U}\,g]$ is valid (see Figure 21). It first computes F and G, characterising the states of M where f and g are valid, and then iteratively computes the least fixed point defined in Equation 7. The loop responsible to compute this least fixed point, is initiated with the variable Q assigned *BddFalse*, which represents the empty set of states (the least value of the set of states lattice), and Q' assigned the image of Q by the predicate transformer. If Q' is different from Q, then the fixed point has not yet been reached, and an iteration is necessary: Q takes the value of Q' and Q' is updated with the image of Q by the predicate transformer. Once Q' is equal to Q, then the fixed point is reached, the loop halts, and the value of Q is returned.

$SmcEG(M, f)$ computes the states of M where $\mathbf{EG}f$ is valid (see Figure 22). It first computes F, the BDD for the set of states of M where f is valid, and then computes the greatest fixed point defined in Equation 6. The algorithm is much similar to that of Figure 21, the main difference being that the initial value of Q is *BddTrue*, that is, the characteristic function of the set of all possible states (the greatest element of the set of states lattice).

Note that in these algorithms, the sets of states might contain both reachable and unreachable states. This occurs most clearly in the *SmcEG* algorithm, where Q is initially assigned *BddTrue*, which is the characteristic function of all possible states, both reachable and unreachable. This also happens in the base cases of the algorithm *SmcAux*, when the CTL formula is a single proposition. In that case, the returned BDD represents a function that characterizes all the states where that proposition is true, be they reachable or unreachable. This is not a problem, however, as traversal of the state space is realized backwards. As discussed in Section 6.1, the inverse image of an unreachable state is unreachable. So the

```
function SmcEU(M: Kripke structure, f: CTL, g: CTL): BDD
variables
  Q, Q', F, G: BDD
begin
  F := SmcAux(M, f);
  G := SmcAux(M, g);
  Q := BddFalse
  Q' := BddOr(G, BddAnd(F, Backward(M, Q)));
  while Q ≠ Q' do
    Q := Q'
    Q' := BddOr(G, BddAnd(F, Backward(M, Q)))
  end while;
  return Q
end function SmcEU
```

Fig. 21. Algorithm for $\mathbf{E}[f\,\mathbf{U}\,g]$ formulae

algorithms guarantee that the set of states that are returned indeed satisfy the given CTL formula.

```
function SmcEG(M: Kripke structure, f: CTL): BDD
variables
  Q, Q', F: BDD
begin
  F := SmcAux(M, f);
  Q := BddTrue
  Q' := BddAnd(F, Backward(M, Q))
  while Q ≠ Q' do
    Q := Q';
    Q' := BddAnd(F, Backward(M, Q))
  end while;
  return Q;
end function SmcEG
```

Fig. 22. Algorithm for $\mathbf{EG}f$ formulae

Example 10 (Symbolic verification of ABPsender). To illustrate the symbolic model-checking algorithm, we apply it to the same verification considered in Example 9: we check that $\mathbf{EG}(\neg s \wedge \neg w)$ is satisfied by $ABPsender$. Now, the verification is performed by the function call $Smc(ABPsender, \mathbf{EG}(\neg s \wedge \neg w))$ (see Figure 19).

Basically, most of the computation is carried out as a result of the function call $SmcEG(ABPsender, \neg s \wedge \neg w)$ (see Figure 22). Table 3 contains a trace for the values of Q, $Backward(ABPsender, Q)$, Q' and $Q = Q'$ during the different iterations of the while statement in the fixed point computation. Actually, the values displayed in this table are that of the boolean formulae represented by Q and Q', instead of the less human-friendly BDDs. The result returned by the

function call is the BDD for $g \wedge \neg w \wedge \neg s$, which is also that of the characteristic function for the set of initial states. Therefore, the symbolic model-checking returns a *true* answer, stating that the formula $\mathbf{EG}\neg s \wedge \neg w$ is valid in the Kripke structure $ABPsender$.

Table 3. Trace of function call $SmcEG(ABPsender, \mathbf{EG}\neg s \wedge \neg w)$

	F	Q	$Backward(Q)$	Q'	$Q = Q'$
Init.	$\neg s \wedge \neg w$	*true*	$(\neg g \wedge w \wedge \neg s) \vee$ $(g \wedge \neg w \wedge \neg s) \vee$ $(\neg g \wedge \neg w \wedge s)$	$g \wedge \neg w \wedge \neg s$	No
Iter. 1	$\neg s \wedge \neg w$	$g \wedge \neg w \wedge \neg s$	$(w \wedge \neg g \wedge \neg s) \vee$ $(g \wedge \neg w \wedge \neg s)$	$g \wedge \neg w \wedge \neg s$	Yes

Exercise 14. Apply the algorithms presented in this section to model check that $ABPsender \models \mathbf{AGEF}g$.

Handling Fairness in Verification. As we have seen in Section 3.4, a fairness constraint restricts the set of paths that are considered as part of the model. Recall that a (weak) fairness constraint κ is characterized by a set of states S_κ that one can specify, for instance, as a temporal logic formula. In that approach, a path is part of the model, and is said to be a *fair path*, if it visits infinitely often S_κ, that is

$$\forall\, i \mid i \geq 0 \bullet \exists\, j \mid j \geq i \bullet \pi(j) \in S_\kappa.$$

In this section, we describe how to apply symbolic methods to perform model checking of CTL formulae under fairness constraints. We first present the case of the **EG** operator and then cover the remaining situations.

Model checking under fairness constraints only takes into account fair paths. Let $\mathcal{K} = \{\, \kappa_1, \ldots \kappa_n \,\}$ be the set of fairness constraints on a Kripke structure M. The property $\mathbf{EG}\varphi$ is valid of state s of M if there is a path π containing s, where φ holds from s onwards, and where each κ_i holds infinitely often. The set C of states where $\mathbf{EG}\varphi$ holds under fairness constraints $\mathcal{K}$ is such that:

1. φ holds in every state $s \in C$;
2. for every constraint κ_i and state $s \in C$, there exists a non-empty path leading to a state $s' \in C$ where κ_i holds, that is:

$$s \models \mathbf{EXE}[\varphi \mathbf{U} \kappa_i \wedge s'].$$

We denote $\mathbf{EG}^{\mathcal{K}}\varphi$ the set of states that satisfy $\mathbf{EG}\varphi$ under the set of fairness constraints $\mathcal{K}$, and characterize it with the following fixed-point-based expression, from which an algorithm can be derived much in the same as we did previously:

$$\mathbf{EG}^{\mathcal{K}}\varphi = \mu\, Z \bullet \varphi \wedge \bigwedge_{\kappa_i \in \mathcal{K}} \mathbf{EXE}[\varphi \mathbf{U} (Z \wedge \kappa_i)]$$

The set of fair states, defined as $F_{\mathcal{K}} = \mathbf{EG}\mathit{true}$, contains all those states belonging to a path that satisfies each constraint infinitely often. Once this set has been computed, model checking can be adapted so that only fair states are considered. (Taking as reference function *SmcAux* of Figure 19, we only need to perform an additional intersection of the set of resulting states with the set of fair states.)

Results and Extensions to Symbolic Model-Checking. Symbolic model-checking has been used to verify a large variety of systems: hardware descriptions [80], software [10], and protocols [186, 64], in particular. The size of the Kripke structures used in these verification has been routinely much larger than 10^{20} states [37].

An extremely useful feature of model checking is the possibility to compute counterexamples (or witnesses) when a universal formula is false (when an existential formula is true) [65]. For instance, the counterexample of an $\mathbf{AG}f$ formula is a path from an initial state to a state where f is not valid.

In practice, symbolic model-checking is well-suited for the verification of the control components of a system, but it performs poorly with data parts. The reason is that BDDs are ill-suited to represent arithmetic expressions or other data-intensive operations. Practically this means that the symbolic model-checking techniques presented in this chapter cannot be used directly to uncover bugs such as the famous Pentium FDIV bug (FDIV being the instruction for floating-point division in the Pentium). This bug occurred in the first generation of Pentium processors and was uncovered in 1994. It caused some division operations to return erroneous results by a small margin, and was due to five uninitialized entries out of a total of 1066 in a lookup table used in the division. The probability that the result be erroneous was 1 out of 9 billions, which would make it very unlikely (or expensive) to find the error by simple simulation and testing methods.

An approach to verify this type of system has been to use data structures other than BDDs to represent the data parts of the system. Word-level model-checking [67] is an example of such an approach; it uses functions mapping boolean vectors into integers to model the system under verification. The internal representation of these functions is a combination of two different classes of data structures: multi-terminal binary decision diagrams (MTBDD) represent the control parts, and binary moment diagrams [35] (BMD) represent the data parts. This technique has indeed been successful in uncovering a design bug in the implementation of division in the floating-point unit of the Pentium processor [67].

Another limitation of symbolic model-checking lies in the expressiveness of the specification logic CTL. Properties asserted in CTL are of a qualitative nature; for instance *if A happens then necessarily B happens in the future*. To express quantitative properties, such as *if A then necessarily B will happen between 4 and 8 time units in the future*, it is necessary to nest several **X** operators into syntactically complex and error prone formulae. One possible solution is to write a preprocessor that converts formulae in a quantitative variant of CTL into an equivalent CTL formula and use the standard symbolic model-checking algorithm

[95]. Another solution is to develop special-purpose algorithms or model representations for this type of formula. Some tools [43, 227] have an even more powerful capability of computing the lower and upper bounds of all possible intervals between two given events. To consider also continuous-time systems it is necessary to develop completely different techniques based on timed automata [8].

6.4 Using NuSMV for Symbolic Model-Checking

NuSMV [59, 58] is a good representative of the possibilities provided by the symbolic model-checking algorithms we have described. It is a reengineering of Carnegie Mellon University SMV, initially developed by McMillan [184], and then extended by Edmund Clarke and his students. McMillan later developed techniques to combine theorem proving techniques and symbolic model-checking, and implemented them in Cadence SMV. The features that we present here are that of NuSMV.

NuSMV (which we will call simply SMV from now on) is an open-source symbolic model-checker for the temporal logics CTL and LTL, and incorporates algorithms for quantitative temporal analysis. It uses the third-party BDD implementation CUDD [243]. Moreover, it also includes some bounded model-checking techniques as presented later on in Section 7.

In the language of SMV, the user describes a design as a collection of modules and temporal properties to be verified. Module instances may be synchronous, in which case all instances take transitions in parallel, or asynchronous, so that a randomly chosen instance takes a transition at each time step. The description of a module contains a list of variables (internal and parameters) and their data type (boolean, scalar, or fixed-length arrays), and an expression, known as transition relation, relating the current and the next values these variables. Intermodule communication is realised by means of shared variables.

Through a command-line interface, the user can load a design, flatten the module hierarchy and then build the symbolic, BDD-based, representation of the corresponding finite-state transition structure. From that point on, the user may either use simulation (deterministic, random, or interactive) to explore the design, or verification to check whether the properties are established or not. If a property is violated, then a counterexample is built and provided to the user. This counterexample can be investigated in simulation mode.

Exercise 15. Access the site of NuSMV at `http://nusmv.irst.itc.it/`. Download the tool and apply it to the provided examples.

7 Bounded Model-Checking

With the techniques we have presented so far, BDD-based symbolic model-checking is practically limited to models with up to a few hundred boolean variables. This makes it unsuitable to directly verify large applications such as complex hardware designs (in which each flip-flop is modelled as a boolean variable) or even simple components of software (where a single integer variable

must be modelled as an array of 32 booleans). For this type of application, one must resort to writing a simplified version of the design in a tool-specific language (such as SPIN or NuSMV) for the sole purpose of verification; this is a time-consuming, error-prone activity requiring specialized knowledge.

A lot of progress has been made in the area of propositional satisfiability (SAT) solving. Today, state-of-the art SAT solvers can handle tens of thousands of variables. Section 7.1 gives an overview of the approach employed in these tools.

These eye-catching results have drawn the attention of the formal verification community which had been working hard to devise algorithmic methods that could scale up several orders of magnitude larger than BDD-based methods. But, SAT solvers are essentially limited to determine if a given proposition formula is satisfiable. So, we needed to find out how the model-checking problem of determining if a Kripke structure is a model for a temporal logic formula can be expressed as a propositional satisfiability problem. We will present an answer to this issue in Section 7.2, following the seminal work of [27].

7.1 Efficient SAT-Solvers

Given a propositional formula $\varphi(x_1, \ldots x_n)$, a SAT-solver returns a model (that is, a valuation of the variables $x_1, \ldots x_n$) of the formula, if it is satisfiable, or says that it is unsatisfiable. This problem has been shown to be NP-complete, however the method proposed by Davis, Putnam, Loveland and Logemann [79, 78] (known as Davis-Putnam or DPLL) has generally been successful in practice.

A *literal* is an atomic variable, or the negation thereof. A *clause* is a disjunction of literals. A clause is said to be a *unit* if it contains one literal. DPLL is based on the assumption that the input is given in clausal form, that is, as a conjunction of clauses, and searches for an assignment to variables that validates each clause. The construction of this search is based on the following two laws of boolean algebras:

$$\begin{array}{cc} \text{(unit resolution)} & x_i \wedge (\neg x_i \vee \varphi) = x_i \wedge \varphi \\ \text{(unit subsumption)} & x_i \wedge (x_i \vee \varphi) = x_i \end{array}$$

Additionally, if x_i is a variable occurring in φ, φ is satisfiable if, and only if, either $\varphi \wedge x_i$ or $\varphi \wedge \neg x_i$ is satisfiable.

A primitive version of the DPLL algorithm is given in Figure 23. This algorithm takes as input a set of clauses S, and returns a boolean to indicate if S is satisfiable, and a set of literals constituting a model for S, in case S is satisfiable.

The function SAT uses an auxiliary function SAT' that basically searches a model (a satisfying assignment of the variables) in a binary decision tree, where each decision is a variable assignment. The function SAT' has a parameter m that stores a partial model, that is the decisions already taken with respect to the variable assignment. Function SAT' first realises the *propagation* of unit clauses (by implementing resolution and subsumption), thus simplifying the set of clauses and the clauses themselves.

When there are no more unit clauses to propagate, a variable is *chosen* among the remaining clause set. The problem is then split into two subproblems, as S is satisfiable if, and only if, one of $S \cup \{ l \}$ and $S \cup \{ \neg l \}$ is satisfiable.

Observe that this algorithm removes from S the literals and the clauses that have been satisfied under the current assignment m. Thus, if S is empty, no more clauses need be satisfied and the original clause set is satisfiable; conversely, if a clause becomes empty, then some clause cannot be satisfied (there is a *conflict*) and the current value of the assignment m cannot be a model for the original clause set. In that second situation, the algorithm backtracks and chooses another variable assignment.

```
function SAT(S: set of clauses): bool, set of literals
begin
   return SAT'(S, ∅)
end function SAT

function SAT'(S: set of clauses, m: set of literals): bool, set of literals
variables
   l: literal; r: bool; m': set of literals;
begin
   while ∃ C ∈ S • C = { l } do (* C is a unit clause *)
      m := m ∪ C;
      remove from S clauses containing l;
      remove ¬l from all clauses of S where it occurs;
      if S = ∅ then return (true, m)
      else if ∅ ∈ S then return (false, −)
      end if
   end while;
   l is a literal occurring in S
   r, m' := SAT'(S ∪ { l }, m);
   if r then return true, m'
   else
      r, m' := SAT'(S ∪ { ¬l }, m);
      if r then return (true, m')
      else return (false, −)
      end if
   end if
end function SAT'
```

Fig. 23. High-level DPLL algorithm for solving propositional satisfiability

Several techniques that significantly improve on this basic algorithm have been proposed. These techniques aim at pruning the search tree visited by SAT' without incurring too much penalty to maintain or compute additional information. One first important modification to the basic algorithm, implemented in the GRASP SAT-solver [178], keeps track of the implications performed during propagation and, in case a conflict is found, uses this information to apply

non-chronological backtracking, that is backtracking several levels at once in the search tree, and therefore obtaining large prunings. A second improvement is to *learn* from such conflicts to infer so called *conflict clauses*. Adding such clauses to the clause set S makes it possible to avoid exploring similar parts of the search tree. To avoid or limit a blow up in the size of S, some tools associate an activity grade to such learnt clauses, and remove them from S when this grade is below a certain threshold.

A second point where the basic algorithm can be refined is in the choice of the literal on which splitting occurs. If the given problem is unsatisfiable, we shall target for a variable assignment that will quickly generate a conflict; on the other hand, if it is satisfiable, then the choice shall try to satisfy as many clauses as possible (to try) to come closer to find a model. Several heuristics based on counting the occurrences of each variables in unresolved clauses have been proposed. The drawback of these methods is that they have to update counters associated to variables during propagation and backtracking, which incurs a significant time penalty which may not be compensated by the reduction in the size of the search space.

Approaches such as *Variable State Independent Decaying Sum* [200] (VSIDS) avoid such problem by maintaining a grade for each literal and branching on the literal with the largest grade. The initial grade is the number of occurrences in the given clause set. It is incremented each time a clause containing the literal is added, and is periodically decayed. Thus, VSIDS tends to choose variables that belong to conflict sets, and avoids updating counters during propagation and backtracking.

Finally, an important design decision is that of the representation for the clause set, which needs to support efficient variable assignment and backtracking. The classic approach consists in representing each clause as a list of literals and to associate to each literal the list of clauses in which it appears. Moreover, each clause maintains counters of the number of satisfied and unsatisfied literals, from which it is straightforward to decide if a clause is satisfied, unsatisfied or is a unit. The drawback of this approach is that the data structures need to be updated during backtracking.

More recently, the technique *watched literals* [200] has been developed by the authors of the Chaff SAT-solver. It is based on the observation that what really matters is that, if a clause has more than one unassigned literal, then it can be neither unsatisfiable nor a unit. Thus each clause only maintains references to two literals that need not be altered when the algorithm backtracks. We point the reader to [259, 174] for further discussions and experiments on modern implementation techniques in propositional SAT-solvers.

7.2 LTL and Bounded Paths

Expressing suitably the model-checking problem as a propositional satisfiability problem is the key to the application of efficient SAT-solvers instead of, or in complement to, BDD-based symbolic algorithms. The bounded model-checking approach consists in, given a specification as a temporal logic formula ϕ and a

Kripke structure model M of the design, looking for a counterexample execution composed of a bounded number of states. We need to introduce the following definitions to ground our discussion on bounded model-checking.

Definition 11 (Universal and existential validity of LTL). *An* LTL *formula* φ *is* universally valid *in a Kripke structure* M*, denoted* $M \models \varphi$*, if, for all paths* π *of* M*,* $M, \pi \models \varphi$*. An* LTL *formula is* existentially valid *in* M*, denoted* $M \models^e \varphi$ *if there is a path* π *such that* $M, \pi \models \varphi$*.*

In Section 4, we have presented model-checking techniques to prove the universal validity of LTL formulae. Clearly, we have that $M \models \varphi$ if, and only if, $M \not\models^e \neg\varphi$, and checking for universal validity of φ amounts to show existential unsatisfiability of $\neg\varphi$. In other words, if we can find a counterexample (that is, show existential validity) of $\neg\varphi$, then the formula φ is not universally valid. Basically, this is the theoretical justification of bounded model-checking. Another aspect of bounded model-checking is related to the "bounded" adjective. The search of the counterexample is restricted to a subset of the state-space: those states that can be reached from the initial states in a bounded number of transitions. Theoretically, the size of the bound can be augmented up to the *diameter* of the transition graph, that is the maximum number of transitions that need to be taken to reach any state. Determining the tightest bound to make bounded model-checking complete is the subject of active research [68].

To illustrate this approach, assume we want to check that the invariant $\mathbf{G}p$ is satisfied by a Kripke structure $M = (S, T, I, L)$ over AP. There is a counterexample of length at most $k + 1$, called a k-bounded counterexample if, and only if, there is some state, reachable in at most k transitions that satisfies p. Let $\mathbf{v}_i$ (respectively, p_i), for $0 \leq i \leq k$, denote the valuation of the atomic propositions in AP (respectively, p) at the i-th state of the path. The existence of a k-bounded counterexample can thus characterized by the satisfiability of the following (propositional) formula:

$$I(\mathbf{v}_0) \wedge \left(\bigwedge_{j=0}^{k-1} T(\mathbf{v}_j, \mathbf{v}_{j+1}) \right) \wedge \left(\bigvee_{j=0}^{k-1} \neg p_j \right)$$

In the remainder of this section, we show how this approach can be generalized to express the existence of a k-bounded counterexample of any LTL formula φ in a Kripke structure M. We first introduce the semantics of LTL for bounded model-checking, and we then show how to express the bounded model-checking problem as a satisfiability problem.

LTL Semantics for Bounded Model-Checking. Bounded model-checking decides the validity of an LTL formula with respect to bounded paths. So, given an LTL formula φ, a Kripke structure M, and a positive number k (the length of the considered path prefixes), we want to express the existence of a path π composed initially of $k - 1$ transitions and k states, where $M, \pi \models \varphi$. Of course, even though we only consider a bounded number of states and transitions, path

π may be an infinite path if there exists some *back loop*, that is, if there is a $l \leq k$ such that $(\pi(k), \pi(l)) \in T$. Actually, only such paths with back loops can be models for formulae of the form $\mathbf{G}\psi$. We need to distinguish the concepts of loop and non-loop paths.

Definition 12 (Loop path). *In a Kripke structure M, a sequence of states $\pi = s_0, s_1, \ldots s_k$ is a (k,l)-*loop *if $\forall\, i \mid 0 \leq i \leq k-1 \bullet (s_i, s_{i+1}) \in T$ and $\exists\, l \mid 0 \leq l \leq k \bullet (s_k, s_l) \in T$. π is a k-*loop*, if there is an l, with $0 \leq l \leq k$, such that π is a (k,l)-loop.*

In case the prefix of a path does not contain a back loop, then nothing can be said about its infinite behaviour. In particular, it is not possible to state that it satisfies an $\mathbf{G}\varphi$ formula. Based on this observation, we introduce the concept of bounded semantics which is used to define bounded validity and bounded model-checking. If the path has a loop, then it is an infinite path and it maintains its usual semantics.

Definition 13 (Bounded semantics for k-loop). *Let π be a k-loop in a Kripke structure M. The* LTL *formula φ is valid with bound k on π, denoted $\pi \models_k \varphi$, if $M, \pi \models \varphi$.*

If the path does not contain a back loop, we have to decide if a given LTL formula φ is valid considering only k consecutive states.

Definition 14 (Bounded semantics for non k-loop). *Let π be a path that is not a k-loop in a Kripke structure M. The* LTL *formula φ is valid with bound k on π, denoted $\pi \models_k \varphi$ iff $M, \pi \models_k^0 \varphi$, as defined by the following rules, where p denotes an atomic proposition, and φ_1 and φ_2 denote* LTL *formulae:*

$\pi \models_k^i p$ *iff* $p \in M.L(\pi(i))$
$\pi \models_k^i \neg p$ *iff* $p \notin M.L(\pi(i))$
$\pi \models_k^i \varphi_1 \wedge \varphi_2$ *iff* $\pi \models_k^i \varphi_1$ *and* $\pi \models_k^i \varphi_2$
$\pi \models_k^i \mathbf{G}\varphi$ *is always false*
$\pi \models_k^i \mathbf{F}\varphi$ *iff* $\exists\, j \mid i \leq j \leq k \bullet \pi \models_k^j \varphi$
$\pi \models_k^i \mathbf{X}\varphi$ *iff* $i < k \wedge \pi \models_k^{i+1} \varphi$
$\pi \models_k^i \varphi_1 \mathbf{U} \varphi_2$ *iff* $\exists\, j \mid i \leq j \leq k \bullet (\pi \models_k^j \varphi_2) \wedge \forall\, l \mid i \leq l < j \bullet (\pi \models_k^l \varphi_1)$
$\pi \models_k^i \varphi_1 \mathbf{W} \varphi_2$ *iff* $\exists\, j \mid i \leq j \leq k \bullet (\pi \models_k^j \varphi_2) \wedge \forall\, l \mid i \leq l < j \bullet (\pi \models_k^l \varphi_1)$

For the **G** and **W** operators, the definition excludes that the formula be valid by means of an infinite path, thus the **W** operator has the same conditions as the **U** operator (recall that $\varphi_1 \mathbf{W} \varphi_2$ is equivalent to $(\varphi_1 \mathbf{U} \varphi_2) \vee (\mathbf{G}\varphi_1)$).

We use the notation $M \models_k \varphi$ to state that there is a path π of M (with or without back loop) such that $\pi \models_k \varphi$. The following theorem states that existential model-checking can be reduced to bounded model-checking (its proof is in [27]).

Theorem 1. *Let φ a normalized* LTL *formula, M a Kripke structure. Then, $M \models^e f$ iff $\exists\, k \geq 0 \bullet M \models_k \varphi$.*

Bounded Model-Checking as Propositional Satisfiability. We now show how bounded model-checking can be reduced to propositional satisfiability. We will consider only LTL formulae in normal negation form (see Section 4.3). Given a normalized formula φ, a Kripke structure M and a bound k, the goal is to construct a propositional formula that is satisfiable if and only if $M \models_k \varphi$. To achieve this, one needs to express the constraints that a path of length k has to satisfy to, first, be a path of M, and, second, be a model for φ. These constraints are expressed as propositional formulae on variables $\mathbf{v}_0$,...$\mathbf{v}_k$ that represent the value of the vectors of state variables at each step of the path. Also, each atomic proposition p of the Kripke structure is instantiated $k + 1$ times: $p_0, \ldots p_k$ to represent the assignment to the corresponding proposition in the corresponding state of the path.

We first define a formula $[\![M]\!]_k$ that relates variables of k consecutive states lying on paths of a Kripke structure M. The first state must be an initial state, thus satisfy the characteristic function I, and each pair of states (s_k, s_{k+1}) must be a transition, thus satisfying the characteristic function T of the transition relation:

$$[\![M]\!]_k = I(\mathbf{v}_0) \wedge \left(\bigwedge_{j=0}^{k-1} T(\mathbf{v}_j, \mathbf{v}_{j+1}) \right)$$

We also define an auxiliary formulae $L_{k,l}$ that identifies the case where there is a transition from the k-th state of a path to the l-th state: $L_{k,l} = T(\mathbf{v}_k, \mathbf{v}_l)$. This notation will be later useful to characterize (k, l)-loops. We also define L_k identifying the situation where there is a transition from the k-th state of the path to a state identified with lower indices: $L_k = \bigvee_{l=0}^{k} L_{k,l}$. Again, this definition will be later used to identify k-loops.

Furthermore, we define auxiliary formulae to state that φ holds on path π, with depth k. Two sub-cases need to be considered. Firstly, the case in which π is not a k-loop; we will write $[\![\varphi]\!]_k^i$ for the formula that states that φ is valid on path π starting at the i-th position and considering up to the k-th state.

Definition 15 (LTL validity on a path without loop). *Let φ be a normalized* LTL *formula, $k \geq 0$ and $i \geq 0$, then $[\![\varphi]\!]_k^i$ is the propositional formula characterising the paths π without back loop such that $\pi \models_k \varphi$. It is defined as follows, where p denotes a propositional variable, p_i denotes the i-th copy of p, and φ_1 and φ_2 denote normalized* LTL *formulae:*

$$[\![p]\!]_k^i = p_i$$
$$[\![\neg p]\!]_k^i = \neg p_i$$
$$[\![\varphi_1 \wedge \varphi_2]\!]_k^i = [\![\varphi_1]\!]_k^i \wedge [\![\varphi_2]\!]_k^i$$
$$[\![\varphi_1 \vee \varphi_2]\!]_k^i = [\![\varphi_1]\!]_k^i \vee [\![\varphi_2]\!]_k^i$$
$$[\![\mathbf{G}\varphi]\!]_k^i = \mathit{false}$$
$$[\![\mathbf{F}\varphi]\!]_k^i = \bigvee_{j=i}^{k} [\![\varphi]\!]_k^j$$
$$[\![\mathbf{X}\varphi]\!]_k^i = \begin{cases} [\![\varphi]\!]_k^{i+1} & \mathit{if}\ i < k \\ \mathit{false} & \mathit{otherwise} \end{cases},$$

$$[\![\varphi_1 \mathbf{U} \varphi_2]\!]_k^i = \bigvee_{j=i}^{k} \left([\![\varphi_2]\!]_k^j \wedge \bigwedge_{l=i}^{j-1} [\![\varphi_1]\!]_k^l \right),$$
$$[\![\varphi_1 \mathbf{W} \varphi_2]\!]_k^i = \bigvee_{j=i}^{k} \left([\![\varphi_2]\!]_k^j \wedge \bigwedge_{l=i}^{j-1} [\![\varphi_1]\!]_k^l \right).$$

Secondly, we consider the case where π is a (k,l)-loop, and we write $[\![\varphi]\!]_{l,k}^i$ for the formula that states that φ is valid on the (k,l) loop π starting at the i-th position.

Definition 16 (LTL validity on a (k,l)-loop). *Let φ a normalized* LTL *formula, and $k \geq l, i \geq 0$. $[\![\varphi]\!]_{k,l}^i$ is the propositional formula characterising the (k,l)-loop π such that $\pi \models_k \varphi$ is defined as follows, where p denotes a propositional variable, φ_1 and φ_2 denote normalized* LTL *formulae:*

$$[\![p]\!]_{k,l}^i = p_i$$
$$[\![\neg p]\!]_{k,l}^i = \neg p_i$$
$$[\![\varphi_1 \wedge \varphi_2]\!]_{k,l}^i = [\![\varphi_1]\!]_{k,l}^i \wedge [\![\varphi_2]\!]_{k,l}^i$$
$$[\![\varphi_1 \vee \varphi_2]\!]_{k,l}^i = [\![\varphi_1]\!]_{k,l}^i \vee [\![\varphi_2]\!]_{k,l}^i$$
$$[\![\mathbf{G}\varphi]\!]_{k,l}^i = \bigwedge_{j=min(i,l)}^{k} [\![\varphi]\!]_{k,l}^j$$
$$[\![\mathbf{F}\varphi]\!]_{k,l}^i = \bigvee_{j=min(i,l)}^{k} [\![\varphi]\!]_{k,l}^j$$
$$[\![\mathbf{X}\varphi]\!]_{k,l}^i = [\![\varphi]\!]_{k,l}^{succ(i)}$$
$$[\![\varphi_1 \mathbf{U} \varphi_2]\!]_{k,l}^i = \bigwedge_{j=i}^{k} \left([\![\varphi_2]\!]_{k,l}^j \wedge \bigvee_{m=i}^{j-1} [\![\varphi_1]\!]_{k,l}^m \right) \vee \bigvee_{j=l}^{i-1} \left([\![\varphi_2]\!]_{k,l}^j \wedge \bigwedge_{m=i}^{k} [\![\varphi_1]\!]_{k,l}^m \wedge \bigwedge_{m=l}^{j-1} [\![\varphi_1]\!]_{k,l}^m \right).$$
$$[\![\varphi_1 \mathbf{W} \varphi_2]\!]_{k,l}^i = \bigwedge_{j=min(i,l)}^{k} ([\![\varphi_1]\!]_{k,l}^j) \vee \bigvee_{j=i}^{k} \left([\![\varphi_2]\!]_{k,l}^j \wedge \bigvee_{m=i}^{j-1} [\![\varphi_1]\!]_{k,l}^m \right) \vee \bigvee_{j=l}^{i-1} \left([\![\varphi_2]\!]_{k,l}^j \wedge \bigwedge_{m=i}^{k} [\![\varphi_1]\!]_{k,l}^m \wedge \bigwedge_{m=l}^{j-1} [\![\varphi_1]\!]_{k,l}^m \right).$$

We have now introduced all the notations required to give the definition of the translation of LTL bounded model-checking as a propositional satisfiability problem.

Definition 17. *Let φ be a normalized* LTL *formula, M a Kripke structure and $k \geq 0$ a bound, then $M \models_k^e \varphi$ if, and only if, the propositional formula $[\![M, \varphi]\!]_k$ on variables $\mathbf{v}_0, \ldots \mathbf{v}_k$ is satisfiable, where:*

$$[\![M, \varphi]\!]_k = [\![M]\!]_k \wedge \left(\left(\neg L_k \wedge [\![\varphi]\!]_k^0 \right) \vee \bigvee_{l=0}^{k} \left(L_{k,l} \wedge [\![\varphi]\!]_{l,k}^0 \right) \right)$$

This definition states that $[\![M, \varphi]\!]_k$ is satisfiable if and only if $\mathbf{v}_0, \ldots \mathbf{v}_k$ are the $k+1$ first states of a path in M (that is, $[\![M]\!]_k$ is satisfiable), and if these same variables form a k-loop where φ is established (then $\bigvee_{l=0}^{k} \left(L_{k,l} \wedge [\![\varphi]\!]_{l,k}^0 \right)$ is satisfiable) or the variables form a non k-loop where φ is established (and $\neg L_k \wedge [\![\varphi]\!]_k^0$ is satisfiable).

8 Applications of Model Checking to Software Design

Producing software has been mainly a human activity, and is therefore prone to error. Even the best testing and inspection practices hit the so-called verification ceiling, and their application cannot guarantee the absence of residual defects. On the other hand, model-checking techniques are fully automatic and guarantee the absence of errors. They have been applied successfully to help produce correct hardware components and protocols. This raises the question whether they can be applied to verify software systems.

We have seen that model-checking techniques apply to finite-state systems, and the corresponding tools have static or bounded input languages. This makes them inconvenient to apply to the verification of software, in particular in the case of object-based systems, which typically feature dynamic creation and destruction of data, as well as unbounded control flow patterns.

Thus, to reduce the gap between an actual software artifact and a tractable analysis using the presented techniques, it is necessary to build a verification model of the software behaviour. This model needs to be faithful with respect to the properties of interest and needs to be amenable to existing model-checking tools. While, it is possible to build such models manually, it is highly undesirable, as it is a new source of errors, and the translation would need to be constantly repeated to accompany the evolution of the implementation.

In consequence, there has been several efforts to provide (semi-)automatic model extraction tools for a variety of model checkers and at different levels of abstraction in the software design process, from formal specification languages, such as CSP-OZ [92], modelling languages such as UML [113, 155, 239], to actual programming languages such as C [19, 122, 111, 54, 69] and JAVA [71, 109, 108]. Model extraction is a combination of abstraction techniques, such as the following:

- *Program slicing* consists in, given a program P, and some elements S of P, producing a program P' that includes only the elements of P that influence directly or indirectly S [124]. Typically S consists of some of the variables of the program that are of interest to establish some important properties. When model checking some property ϕ on S, it is sufficient to consider the slice of P with respect to ϕ.
- *Data abstraction* by abstract interpretation makes it possible to soundly substitute rich data types with simpler ones [75]. For example, if a program uses a vector to store a set of elements, and the only relevant information is that some value is in the set, then the vector might be abstracted to an enumeration type with two elements $\{ InSet, NotInSet \}$.
- *Predicate abstraction* abstracts a program P by a boolean program P_B [104]. A boolean program is a possibly non-deterministic program where all variables have the boolean data type. The variables of P_B represent the control-flow predicates of P, and the instructions of P_B model the effect of the instructions of P on these predicates. For instance, suppose P contains a control-flow predicate $x \geq y$, which is abstracted to a boolean variable v

in P_B. Suppose that P contains the assignment $x := x - 1$: how can it be translated as a boolean program on v?
If v is true before the assignment (that is, $x \geq y$), then, afterwards, either v is true (in case $x > y$), or false (in case $x = y$). If v is false before the assignment (that is, $x < y$), then it certainly is false afterwards, as the modeled instruction decreases the value of x. So in the boolean program, the counterpart of the assignment would be if v then $v := \{ \mathit{true}, \mathit{false} \}$, where the expression $\{e_1, e_2\}$ evaluates non-deterministically to e_1 or to e_2. We have presented BDDs and SAT-solvers that are efficient techniques to handle (finite) boolean programs.

Note that in the case of data abstraction and predicate abstraction, the extracted model might be too coarse and model checking can yield false counterexamples. These counterexamples can be used to refine the abstraction, by introducing additional, relevant details in the model [18].

Another, more recent, line of research consists in devising specification logics and model formalisms that are more adequate to support object-based software features, such as allocation and deallocation, evolution of the heap structure, and multiple threads, than Kripke structure and Büchi automata. Although this line of work seems promising, further investigation is needed to show how effective it is, and how it can be combined with other techniques [82].

9 Conclusions

Several books have been published on the subject of model checking: [141] presents an automata-based approach to model checking that underpins his verification tool COSPAN; [66] is a thorough overview of classic model-checking techniques with an emphasis on CTL model-checking; [24] describes the basis of model checking: transition systems as a model of systems, temporal logic as language for requirements, and model checking as verification algorithms.

Elementary Probability Theory

In probability theory [105, for example], an *event* is a subset of some given *sample space*: the event is said to have occurred if the sampled value is in that set. We call elements of the sample space *points*.

A *probability distribution* over the sample space is a function from its events into the closed interval $[0, 1]$, giving for each event the probability of its occurrence. In the general case, for technical reasons, not necessarily all subsets of the sample space are events; but in our case we do take every subset of the sample space to be an event, and so we can regard a probability distribution more simply as a function from points of the sample space (rather than subsets) directly to probabilities. The probability of a set of points (an event) is then just the sum of the probabilities of the points individually; and the probability assigned to the whole sample space—that is, to all points at once—is one.

A *random variable* is a function from points to reals; here, we restrict to nonnegative reals. The *expected value* of a random variable is defined in terms of the probability distribution: it is the "weighted average" obtained by summing over all points the product of the random variable's value at the point and the probability assigned to the point by the distribution. A fact which we use often is that the expected value (over a distribution) of the characteristic function of an event is just the probability that the event will occur (as determined by that distribution).

For example, the sample space for a coin-flip is the set $\{H, T\}$, and a probability distribution for a fair coin over that sample space might be the function $\{H \mapsto 1/2, T \mapsto 1/2\}$. On the other hand, $\{H \mapsto 2/3, T \mapsto 1/3\}$ describes a heads-biased coin.

The function $\{H \mapsto 2, T \mapsto 3\}$ is a random variable over our state space, which could describe many things including, for example, how much money we win on each occasion if H (respectively T) should turn up: \$2 (respectively \$3). The expected value of that random variable over the biased-coin distribution gives how much on average we would win per coin-flip: it is $2 * 2/3 + 3 * 1/3$, that is \$2.33 per flip. On the other hand, if we took the characteristic function $\{H \mapsto 1, T \mapsto 0\}$ of the set $\{H\}$—which we might write $[H]$—then the expected value would be $1 * 2/3 + 0 * 1/3$, that is 2/3, just the probability of getting H.

For us, programs take initial states to final distributions of state, or to sets of distributions if the program contains nondeterminism, and our post-expectations are random variables over the state. Thus a coin-flip program $c := H \,{}_p\!\oplus\, T$ would take any initial state in the program's state-space (also the sample space) to a final distribution $\{H \mapsto p, T \mapsto 1-p\}$ over that state space.

Proofs of Lemmas and Theorems in the UTP

Lemma 1 (export-precondition).

$$(P \vdash Q) = (P \vdash P \wedge Q)$$

Proof.

$$\begin{array}{ll} & P \vdash Q & \textit{[design]} \\ = & \mathit{okay} \wedge P \Rightarrow \mathit{okay}' \wedge Q & \textit{[propositional calculus]} \\ = & \mathit{okay} \wedge P \Rightarrow \mathit{okay}' \wedge P \wedge Q & \textit{[design]} \\ = & P \vdash P \wedge Q & \Box \end{array}$$

Lemma 2 (design-abort). *When a design has not started ($\neg$ okay), it offers no guarantees.*

$$(P \vdash Q)[\mathit{false}/\mathit{okay}] = \mathbf{true}$$

Proof.

$$\begin{array}{ll} & (P \vdash Q)[\mathit{false}/\mathit{okay}] & \textit{[design]} \\ = & (\mathit{okay} \wedge P \Rightarrow \mathit{okay}' \wedge Q)[\mathit{false}/\mathit{okay}] & \textit{[substitution]} \\ = & \mathbf{false} \wedge P \Rightarrow \mathit{okay}' \wedge Q & \textit{[propositional calculus]} \\ = & \mathbf{true} & \Box \end{array}$$

Lemma 3 (condition-right-unit). *Abort is a right-unit for conditions.*

$$p \mathbin{;} \mathbf{true} = p$$

Proof.

$$\begin{array}{ll} & p \mathbin{;} \mathbf{true} & \textit{[sequence]} \\ = & \exists\, v_0 \bullet p[v_0/v'] \wedge \mathbf{true}[v_0/v] & \textit{[v not free in } \mathbf{true}\textit{]} \\ = & \exists\, v_0 \bullet p[v_0/v'] \wedge \mathbf{true} & \textit{[unit for conjunction]} \\ = & \exists\, v_0 \bullet p[v_0/v'] & \textit{[v' not free in p]} \\ = & \exists\, v_0 \bullet p & \textit{[v_0 not free in p]} \\ = & p & \Box \end{array}$$

Lemma 4 (abort-condition).

$$\mathbf{true} \mathbin{;} p \;=\; \exists\, v \bullet p$$

Proof.

$$
\begin{array}{ll}
\mathbf{true}\ ;\ p & \textit{[sequence]} \\
= \exists\, v_0 \bullet \mathbf{true} \wedge p[v_0/v] & \textit{[unit for conjunction]} \\
= \exists\, v_0 \bullet p[v_0/v] & \textit{[}v_0\textit{ not free in }p\textit{, predicate calculus]} \\
= \exists\, v \bullet p & \square
\end{array}
$$

Lemma 5 (not-design).

$$\neg\,(P \vdash Q) = (okay \wedge P \wedge (okay' \Rightarrow \neg\, Q))$$

Proof.

$$
\begin{array}{ll}
\quad \neg\,(P \vdash Q) & \textit{[design]} \\
= \neg\,(okay \wedge P \Rightarrow okay' \wedge Q) & \textit{[propositional calculus]} \\
= okay \wedge P \wedge (okay' \Rightarrow \neg\, Q) & \square
\end{array}
$$

Lemma 6 (*H1*-left-zero).

$$\mathbf{true}\ ;\ P = \mathbf{true}$$

Proof.

$$
\begin{array}{ll}
\quad \mathbf{true}\ ;\ P & \textit{[assumption (}P\textit{ is }\boldsymbol{H1}\textit{)]} \\
= \mathbf{true}\ ;\ (\ okay \Rightarrow P\ ) & \textit{[relational calculus]} \\
= \mathbf{true}\ ;\ (\neg\ okay \vee P\ ) & \textit{[relational calculus]} \\
= (\ \mathbf{true}\ ;\ \neg\ okay\ ) \vee (\ \mathbf{true}\ ;\ P\ ) & \textit{[abort-boolean]} \\
= \mathbf{true} \vee (\ \mathbf{true}\ ;\ P\ ) & \textit{[disjunction-left-zero]} \\
= \mathbf{true} & \square
\end{array}
$$

Lemma 7 (*H1*-left-unit). *Suppose that P is **H1**-healthy.*

$$\mathbb{I}_D\ ;\ P\ =\ P$$

Proof.

$$
\begin{array}{ll}
\quad \mathbb{I}_D\ ;\ P & \textit{[}\mathbb{I}_D\textit{]} \\
= (\ \mathbf{true} \vdash \mathbb{I}_D\ )\ ;\ P & \textit{[design]} \\
= (\ okay \Rightarrow okay' \wedge \mathbb{I}\ )\ ;\ P & \textit{[relational calculus]} \\
= (\ \neg\ okay\ ;\ P\ ) \vee (\ okay \wedge P\ ) & \textit{[condition-right-unit]} \\
= (\ \neg\ okay\ ;\ \mathbf{true}\ ;\ P\ ) \vee (\ okay \wedge P\ ) & \textit{[assumption (}P\textit{ is }\boldsymbol{H1}\textit{), }\boldsymbol{H1}\textit{-left-zero]} \\
= (\ \neg\ okay\ ;\ \mathbf{true}\ ) \vee (\ okay \wedge P\ ) & \textit{[condition-right-unit]} \\
= \neg\ okay \vee (\ okay \wedge P\ ) & \textit{[relational calculus]} \\
= okay \Rightarrow P & \textit{[assumption (}P\textit{ is }\boldsymbol{H1}\textit{)]} \\
= P & \square
\end{array}
$$

Lemma 8 (left-unit-zero-*H1*). *Suppose that P has a left unit and a left zero.*

$$P \;=\; (okay \Rightarrow P)$$

Proof.

$$\begin{array}{ll}
& P & \textit{[assumption ($\mathbb{I}_D$ is left-unit)]}\\
= & \mathbb{I}_D \,;\, P & \textit{[$\mathbb{I}_D$]}\\
= & (\,\mathbf{true} \vdash \mathbb{I}\,) \,;\, P & \textit{[design]}\\
= & (\,okay \Rightarrow okay' \wedge \mathbb{I}\,) \,;\, P & \textit{[relational calculus]}\\
= & (\,\neg\, okay \,;\, P\,) \vee (\,\mathbb{I} \,;\, P\,) & \textit{[condition-right-unit]}\\
= & (\,\neg\, okay \,;\, \mathbf{true} \,;\, P\,) \vee (\,\mathbb{I} \,;\, P\,) & \textit{[assumption (}\mathbf{true}\textit{ is left-zero)]}\\
= & \neg\, okay \vee (\,\mathbb{I} \,;\, P\,) & \textit{[$\mathbb{I}$ is left-unit for relations]}\\
= & \neg\, okay \vee P & \textit{[propositional calculus]}\\
= & okay \Rightarrow P & \Box
\end{array}$$

Lemma 9 (J-split). *For all relations with $okay$ and $okay'$ in their alphabet,*

$$P \,;\, J \;=\; P^f \vee (P^t \wedge okay')$$

Proof.

$$\begin{array}{ll}
& P \,;\, J & \textit{[J, with $v \in \alpha P \setminus \{\, okay \,\}$]}\\
= & P \,;\, (okay \Rightarrow okay') \wedge v' = v & \textit{[propositional calculus]}\\
= & P \,;\, (okay \Rightarrow okay \wedge okay') \wedge v' = v & \textit{[propositional calculus]}\\
= & P \,;\, (\neg\, okay \vee okay \wedge okay') \wedge v' = v & \textit{[relational calculus]}\\
= & P \,;\, \neg\, okay \wedge v' = v \vee (P \,;\, okay \wedge v' = v) \wedge okay' & \textit{[right-one-point, twice]}\\
= & P^f \vee (P^t \wedge okay') & \Box
\end{array}$$

Theorem 1 (*H2* equivalence). *There are two equivalent ways of characterising **H2**-healthy relations.*

$$(P = P \,;\, J) \;=\; [\,P^f \Rightarrow P^t\,]$$

Proof.

$$\begin{array}{ll}
& (P = P \,;\, J) & \textit{[J-split]}\\
= & (P = P^f \vee (P^t \wedge okay')) & \textit{[$okay'$ split]}\\
= & (P = P^f \vee (P^t \wedge okay'))^f \wedge (P = P^f \vee (P^t \wedge okay'))^t & \textit{[substitution]}\\
= & (P^f = P^f \vee (P^t \wedge \mathbf{false})) \wedge (P^t = P^f \vee (P^t \wedge \mathbf{true})) & \textit{[propositional calculus]}\\
= & (P^f = P^f) \wedge (P^t = P^f \vee P^t) & \textit{[reflection]}\\
= & (P^t = P^f \vee P^t) & \textit{[equality (mutual refinement) of programs, propositional calculus]}\\
= & [\,P^f \Rightarrow P^t\,] & \Box
\end{array}$$

Lemma 10 (J is *H2*). *J is **H2**-healthy.*

$$J = \mathbf{H2}(J)$$

Proof.

$$\begin{aligned}
& \mathbf{H2}(J) && \text{[\textbf{H2} and J-split]}\\
&= J^f \vee (J^t \wedge okay') && \text{[J]}\\
&= (\neg\, okay \wedge \mathit{ll}_{rel}^{-okay}) \vee (okay' \wedge \mathit{ll}_{rel}^{-okay}) && \text{[propositional calculus]}\\
&= (\neg\, okay \vee okay') \wedge \mathit{ll}_{rel}^{-okay} && \text{[propositional calculus]}\\
&= (okay \Rightarrow okay') \wedge \mathit{ll}_{rel}^{-okay} && \text{[J]}\\
&= J && \square
\end{aligned}$$

Lemma 11 (*H1*-*H2* is a design). *If P is a relation that is both **H1** and **H2** healthy, then it can be expressed as the design $\neg\, P^f \vdash P^t$.*

Proof.

$$\begin{aligned}
& P && \text{[assumption: P is \textbf{H1}]}\\
&= okay \Rightarrow P && \text{[assumption: P is \textbf{H2}]}\\
&= okay \Rightarrow P\ ;\ J && \text{[J-splitting]}\\
&= okay \Rightarrow P^f \vee (P^t \wedge okay') && \text{[propositional calculus]}\\
&= okay \wedge \neg\, P^f \Rightarrow okay' \wedge P^t && \text{[design]}\\
&= \neg\, P^f \vdash P^t && \square
\end{aligned}$$

Lemma 12 (Designs are *H2*). *For all predicates P and Q that do not have $okay$ and $okay'$ in their alphabets,*

$$[\,(\,P \vdash Q\,)^f \Rightarrow (\,P \vdash Q\,)^t\,]$$

Proof.

$$\begin{aligned}
& (\,P \vdash Q\,)^f && \text{[design]}\\
&= (\,okay \wedge P \Rightarrow \mathbf{false}\,) && \text{[propositional calculus]}\\
&\Rightarrow (\,okay \wedge P \Rightarrow Q\,) && \text{[design]}\\
&= (\,P \vdash Q\,)^t && \square
\end{aligned}$$

Lemma 13 (external-choice-diverge). *Provided P and Q are **R1**-healthy,*

$$(P \,\Box\, Q)^f_f = (P^f_f \vee Q^f_f) \lhd okay \rhd (P^f_f \wedge Q^f_f)$$

Proof.

$$(P \,\Box\, Q)^f_f \qquad \text{[external-choice]}$$

$= (\boldsymbol{CSP2}(P \wedge Q \lhd STOP \rhd P \vee Q))^f_f$ *[**CSP2**-diverge]*

$= (P \wedge Q \lhd STOP \rhd P \vee Q)^f_f$ *[substitution]*

$= (P \wedge Q)^f_f \lhd STOP^f_f \rhd (P \vee Q)^f_f$ *[conditional]*

$= (P^f_f \wedge Q^f_f \wedge STOP^f_f) \vee ((P^f_f \vee Q^f_f) \wedge \neg\, STOP^f_f)$ *[$STOP$-diverge]*

$= (P^f_f \wedge Q^f_f \wedge \boldsymbol{R1}(\neg\, okay)) \vee ((P^f_f \vee Q^f_f) \wedge \neg\, (\boldsymbol{R1}(\neg\, okay)))$

[assumption: P and Q are $\boldsymbol{R1}$-healthy]

$= (P^f_f \wedge (\boldsymbol{R1}(Q))^f_f \wedge \boldsymbol{R1}(\neg\, okay)) \vee$ *[$\boldsymbol{R1}$-wait, $\boldsymbol{R1}$-okay′, twice]*
$(((\boldsymbol{R1}(P))^f_f \vee (\boldsymbol{R1}(Q))^f_f) \wedge \neg\, (\boldsymbol{R1}(\neg\, okay)))$

$= (P^f_f \wedge \boldsymbol{R1}(Q^f_f) \wedge \boldsymbol{R1}(\neg\, okay))$ *[$\boldsymbol{R1}$-extends-over-and, $\boldsymbol{R1}$-disjunctive]*
$\vee$
$((\boldsymbol{R1}(P^f_f) \vee \boldsymbol{R1}(Q^f_f)) \wedge \neg\, (\boldsymbol{R1}(\neg\, okay)))$

$= (P^f_f \wedge \boldsymbol{R1}(Q^f_f) \wedge \neg\, okay) \vee (\boldsymbol{R1}(P^f_f \vee Q^f_f) \wedge \neg\, (\boldsymbol{R1}(\neg\, okay)))$

[$\boldsymbol{R1}$-wait, $\boldsymbol{R1}$-okay′, $\boldsymbol{R1}$-and-not-$\boldsymbol{R1}$, propositional calculus]

$= (P^f_f \wedge (\boldsymbol{R1}(Q))^f_f \wedge \neg\, okay) \vee \boldsymbol{R1}((P^f_f \vee Q^f_f) \wedge okay)$

[assumption: Q is $\boldsymbol{R1}$-healthy, $\boldsymbol{R1}$-extends-over-and]

$= (P^f_f \wedge Q^f_f \wedge \neg\, okay) \vee \boldsymbol{R1}(P^f_f \vee Q^f_f) \wedge okay$

[$\boldsymbol{R1}$-disjunctive, $\boldsymbol{R1}$-wait, $\boldsymbol{R1}$-okay′, assumption: P and Q are $\boldsymbol{R1}$-healthy]

$= (P^f_f \wedge Q^f_f \wedge \neg\, okay) \vee ((P^f_f \vee Q^f_f) \wedge okay)$ *[conditional]*

$= (P^f_f \vee Q^f_f) \lhd okay \rhd (P^f_f \wedge Q^f_f)$ □

Lemma 14 (external-choice-precondition).

$$(\neg\, (P \,\Box\, Q)^f_f \vdash R) \;=\; (\neg\, (P^f_f \vee Q^f_f) \vdash R)$$

Proof.

$\neg\, (P \,\Box\, Q)^f_f \vdash R$ *[design]*

$= okay \wedge \neg\, (P \,\Box\, Q)^f_f \Rightarrow okay' \wedge R$ *[external-choice-diverge]*

$= okay \wedge \neg\, ((P^f_f \vee Q^f_f) \lhd okay \rhd (P^f_f \wedge Q^f_f)) \Rightarrow okay' \wedge R$ *[not-conditional]*

$= okay \wedge (\neg\, (P^f_f \vee Q^f_f) \lhd okay \rhd \neg\, (P^f_f \wedge Q^f_f)) \Rightarrow okay' \wedge R$

[known-condition]

$= okay \wedge \neg\, (P^f_f \vee Q^f_f) \Rightarrow okay' \wedge R$ *[design]*

$= \neg\, (P^f_f \vee Q^f_f) \vdash R$ □

Lemma 15 (external-choice-converge).

$$\begin{aligned}(P \,\Box\, Q)^t_f \;=\;& (P \wedge Q) \lhd STOP \rhd (P \vee Q)^t_f \\ & \vee \\ & (P \wedge Q) \lhd STOP \rhd (P \vee Q)^f_f\end{aligned}$$

Proof.

$$
\begin{aligned}
& (P \,\Box\, Q)^t_f && \textit{[external choice]} \\
= {} & (\boldsymbol{CSP2}(P \wedge Q \lhd STOP \rhd P \vee Q))^t_f && \textit{[}\boldsymbol{CSP2}\textit{-converge]} \\
= {} & (P \wedge Q \lhd STOP \rhd P \vee Q)^t_f \vee (P \wedge Q \lhd STOP \rhd P \vee Q)^f_f && \Box
\end{aligned}
$$

Lemma 16 (design-external-choice-lemma).

$$
\begin{aligned}
& (\neg\, (P \,\Box\, Q)^f_f \vdash (P \,\Box\, Q)^t_f) \;= \\
& \quad ((\neg\, P^f_f \wedge Q^f_f) \vdash ((P^t_f \wedge Q^t_f) \lhd tr' = tr \wedge wait' \rhd (P^t_f \vee Q^t_f)))
\end{aligned}
$$

Proof.

$$
\begin{aligned}
& \neg\, (P \,\Box\, Q)^f_f \vdash (P \,\Box\, Q)^t_f \\
& \qquad \textit{[external-choice-diverge, design, known-condition, propositional calculus]} \\
= {} & (\neg\, P^f_f \wedge \neg\, Q^f_f) \vdash (P \,\Box\, Q)^t_f && \textit{[external-choice-converge]} \\
= {} & (\neg\, P^f_f \wedge \neg\, Q^f_f) \vdash (P \wedge Q \lhd STOP \rhd P \vee Q)^t_f \vee \\
& \qquad\qquad (P \wedge Q \lhd STOP \rhd P \vee Q)^f_f \\
& \qquad\qquad\qquad \textit{[design, propositional calculus]} \\
= {} & (\neg\, P^f_f \wedge \neg\, Q^f_f) \vdash (P \wedge Q \lhd STOP \rhd P \vee Q)^t_f \vee && \textit{[substitution]} \\
& (\neg\, P^f_f \wedge \neg\, Q^f_f) \vdash (P \wedge Q \lhd STOP \rhd P \vee Q)^f_f \\
= {} & (\neg\, P^f_f \wedge \neg\, Q^f_f) \vdash (P \wedge Q \lhd STOP \rhd P \vee Q)^t_f \vee \\
& (\neg\, P^f_f \wedge \neg\, Q^f_f) \vdash (P^f_f \wedge Q^f_f \lhd STOP^f_f \rhd P^f_f \vee Q^f_f) \\
& \qquad\qquad\qquad \textit{[design, propositional calculus]} \\
= {} & (\neg\, P^f_f \wedge \neg\, Q^f_f) \vdash (P \wedge Q \lhd STOP \rhd P \vee Q)^t_f \vee && \textit{[design-post-or]} \\
& (\neg\, P^f_f \wedge \neg\, Q^f_f) \vdash \mathbf{false} \\
= {} & (\neg\, P^f_f \wedge \neg\, Q^f_f) \vdash (P \wedge Q \lhd STOP \rhd P \vee Q)^t_f \vee \mathbf{false} \\
& \qquad\qquad\qquad \textit{[propositional calculus]} \\
= {} & (\neg\, P^f_f \wedge \neg\, Q^f_f) \vdash (P \wedge Q \lhd STOP \rhd P \vee Q)^t_f && \textit{[substitution]} \\
= {} & (\neg\, P^f_f \wedge \neg\, Q^f_f) \vdash (P^t_f \wedge Q^t_f \lhd STOP^t_f \rhd P^t_f \vee Q^t_f) && \textit{[}STOP\textit{-converge]} \\
= {} & (\neg\, P^f_f \wedge \neg\, Q^f_f) \vdash (P^t_f \wedge Q^t_f \lhd \boldsymbol{CSP1}(tr' = tr \wedge wait') \rhd P^t_f \vee Q^t_f) \\
& \qquad \textit{[design-post-conditional-}\boldsymbol{CSP1}\textit{, assumption: } P \textit{ and } Q \textit{ } \mathbf{R1}\textit{-healthy]} \\
= {} & (\neg\, P^f_f \wedge \neg\, Q^f_f) \vdash (P^t_f \wedge Q^t_f \lhd tr' = tr \wedge wait' \rhd P^t_f \vee Q^t_f) && \Box
\end{aligned}
$$

Library Block Specifications

The semantics of the Z functions for library blocks used in the specification of the Ada function stubs are given here. First some auxiliary definitions are given.

*min*16 is the smallest signed 16-bit word (x**y is "x to the power y" with ** defined in the Z toolkit extension of the compliance tool).

*max*16 is the largest signed 16-bit word.

z

$$min16 \mathrel{\widehat=} \sim(2 ** 15)$$

z

$$max16 \mathrel{\widehat=} (2 ** 15) - 1$$

The function *wrange* limits its input to a signed 16-bit word.

z

$$wrange : \mathbb{Z} \to \mathbb{Z}$$

$$\forall x : \mathbb{Z} \bullet$$
$$(x < min16 \Rightarrow wrange(x) = min16) \land$$
$$(min16 \leq x \leq max16 \Rightarrow wrange(x) = x) \land$$
$$(max16 < x \Rightarrow wrange(x) = max16)$$

*min*32 is the smallest signed 32-bit longword.

z

$$min32 \mathrel{\widehat=} \sim(2 ** 31)$$

*max*32 is the largest signed 32-bit longword.

z

$$max32 \mathrel{\widehat=} (2 ** 31) - 1$$

The function *lrange* limits its input to a signed 32-bit longword.

z

$$lrange : \mathbb{Z} \to \mathbb{Z}$$

$$\forall x : \mathbb{Z} \bullet$$
$$(x < min32 \Rightarrow lrange(x) = min32) \land$$
$$(min32 \leq x \leq max32 \Rightarrow lrange(x) = x) \land$$
$$(max32 < x \Rightarrow lrange(x) = max32)$$

The function *scale* multiplies x and y and divides the result by z, limiting the result to a word range. If the denominator is zero, the result is limited to the smallest or largest word depending on the sign of multiplying x and y.

z

$$\forall\, x,\, y,\, z : \mathbb{Z} \bullet \\ (z \neq 0 \Rightarrow scale(x,\, y,\, z) = wrange((x * y)\ intdiv\ z)) \land \\ (z = 0 \land x * y < 0 \Rightarrow scale(x,\, y,\, z) = min16) \land \\ (z = 0 \land x * y \geq 0 \Rightarrow scale(x,\, y,\, z) = max16)$$

The function *differentiator* calculates the differential (rate of change) by multiplying the difference in x and y (current and previous input) by z (the gain).

z

$$\forall\, x,\, y,\, z : \mathbb{Z} \bullet differentiator(x,\, y,\, z) = wrange(z * (x - y))$$

The function *integrator* performs rectangular integration by adding the previous integral x to the new contribution formed from the current input y and the gain z.

z

$$\forall\, x,\, y,\, z : \mathbb{Z} \bullet integrator(x,\, y,\, z) = lrange(x + 2 * z * y)$$

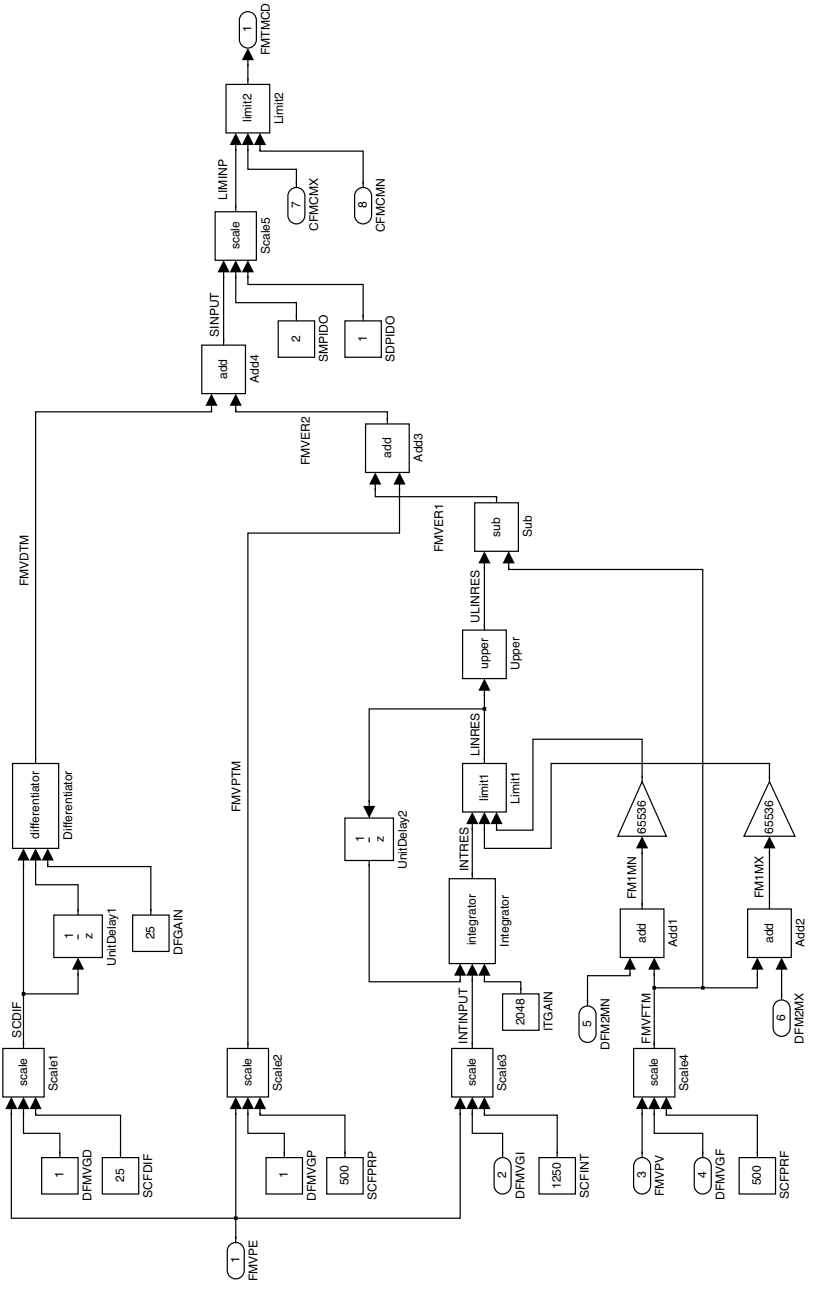


Fig. 1. General PID controller

References

1. M. Abadi and L. Lamport. The existence of refinement mapping. *Theoretical Computer Science*, 83(2):253–284, 1991.
2. M. Abadi and L. Lamport. An old-fashioned recipe for real-time. In J. W. de Bakker, C. Huizing, W. P. de Rover, and G. Rozenberg, editors, *Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.
3. J-R. Abrial. *The B-Book: Assigning Progams to Meanings*. Cambridge University Press, 1996.
4. J-R. Abrial, E. Börger, and H. Langmaack. *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, volume 1165 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
5. Accellera. Property Specification Language Reference Manual, Version 1.1, 2004.
6. B. Alpern and F. B. Schneider. Verifying temporal properties without temporal logic. *ACM Transactions on Programming Languages and Systems*, 11(1):147–167, 1989.
7. A. Alur and T. A. Henzinger. Real-time logics: complexity and expressiveness. In *5th IEEE Symposium on Logic in Computer Science*, page 390–401. IEEE Computer Society, 1990.
8. R. Alur, C. Courcoubetis, and D. L. Dill. Model checking for real time systems. In *5th IEEE Symposium on Logic in Computer Science*, page 414–425. IEEE Computer Society Press, 1990.
9. R. Alur and D. L. Dill. Automata for modelling real-time systems. In M. S. Paterson, editor, *ICALP 90: Automata, Languages and Programming*, volume 443 of *Lecture Notes in Computer Science*, page 322–335. Springer-Verlag, 1990.
10. R. J. Anderson, P. Beame, S. Burns, W. Chan, F. Modugno, D. Notkin, and R. Reese. Model Checking Large Software Specifications. In *4th Symposium on the Foundations of Software Engineering*, page 156–166, 1996.
11. R. Arthan, P. Caseley, C. O'Halloran, and A. Smith. ClawZ: Control laws in Z. In *3rd International Conference on Formal Engineering Methods*, page 169–176. IEEE Press, 2000.
12. N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling. Technical Report RTRG/92/120, Department of Computer Science, University of York, 1992.
13. A. Avižienis. Fault-Tolerant Systems. *IEEE Transactions on Software Engineering*, C-25(12):1304–1312, 1976.
14. R. J. R. Back. Procedural Abstraction in the Refinement Calculus. Technical report, Department of Computer Science, Åbo, Finland, 1987. Ser. A No. 55.
15. R. J. R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.
16. R. J. R. Back and J. Wright. Duality in Specification Languages: A Lattice-theoretical Approach. *Acta Informatica*, 27(7):583 –625, 1990.
17. R. J. R. Back and J. Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.
18. T. Ball, B. Cook, S. Das, and S. K. Rajamani. Refining approximations in software predicate abstraction. In *10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 2988 in Lecture Notes in Computer Science, page 388–403. Springer-Verlag, 2004.

19. T. Ball and S. K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In *SPIN 2001 Workshop on Model Checking of Software*, number 2057 in Lecture Notes in Computer Science, page 103–122. Springer-Verlag, 2001.
20. A. Banerjee and D. Naumann. Representation independence, confinement and access control. In *Principles of Programming Languages*, page 166–177, 2002.
21. J. Barnes. *High Integrity Ada: The Spark Approach*. Addison-Wesley, 1997.
22. G. Behrmann, A. David, and K. G. Larsen. A Tutorial on UPPAAL. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems*, number 3185 in Lecture Notes in Computer Science, page 200–236. Springer-Verlag, 2004.
23. J. Bengtsson and W. Yi. Timed Automata: Semantics, Algorithms, and Tools. In *Lecture Notes on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.
24. B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen, and P. McKenzie. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer Verlag, 2001.
25. G. Bernat, A. Burns, and A. Welling. Portable worst-case execution time analysis using Java byte code. In *6th Euromicro Conference on Real-Time Systems*, 2000.
26. G. Bernat, A. Burns, and A. Wellings. JAVELIN: Worst-case execution time analysis using an architectural neutral form. Technical report, Progress Report on DERA Contract No. CSM/1254, 2000.
27. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, page 193–207. Springer-Verlag, 1999.
28. R. Bird and O. de Moor. *Algebra of Programming*. Prentice-Hall, 1997.
29. N. S. Bjørner, Z. Manna, H. B. Sipma, and T. E. Uribe. Deductive verification of real-time systems using STeP. In M. Bertran and T. Rus, editors, *Transformation-Based Reactive Systems Development*, volume 1231 of *Lecture Notes in Computer Science*, page 21–43. Springer-Verlag, 1997.
30. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety-Critical Software. In *ACM SIGPLAN SIGSOFT Conference on Programming Language Design and Implementation*, page 196–207, 2003.
31. H. J. Boom. A Weaker Precondition for Loops. *ACM Transactions on Programming Languages and Systems*, 4(4):668–677, 1982.
32. P. H. M. Borba, A. C. A. Sampaio, A. L. C. Cavalcanti, and M. L. Cornélio. Algebraic Reasoning for Object-Oriented Programming. *Science of Computer Programming*, 52:53–100, 2004.
33. P. H. M. Borba, A. C. A. Sampaio, and M. L. Cornélio. A Refinement Algebra for Object-oriented Programming. In Luca Cardelli, editor, *European Conference on Object-oriented Programming 2003*, volume 2743 of *Lecture Notes in Computer Science*, page 457–482. Springer-Verlag, 2003.
34. R. E. Bryant. Graph-based algorithm for boolean function manipulation. *IEEE Transactions Computers*, C(35):1035–1044, 1986.
35. R. E. Bryant and Y.-A. Chen. Verification of Arithmetic Circuits with Binary Moments Diagrams. In *32nd ACM/IEEE Design Automation Conference*, page 535–541, 1995.

36. J. R. Büchi. On a Decision Method in Restricted Second Order Arithmetic. In E. Nagel, E. Suppes, and A. Tarski, editors, *International Congress on Logic, Method and Philosophy of Science 1960*, page 1–12. Stanford University Press, 1962.
37. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic Model Checking: 10^{20} states and beyond. In *5th IEEE Symposium on Logic in Computer Science*, page 428–439. IEEE Computer Society Press, 1990.
38. A. Burns, R. Davis, and S. Punnekkat. Feasibility Analysis of Fault-Tolerant Real-Time Task Sets. In *Euromicro Workshop on Real-Time Systems*, page 29–33, 1996.
39. A. Burns, J. Hooman, M. Jospeh, Z. Liu, K. Ramamritham, H. Schepers, S. Schneider, and A. J. Wellings. *Real-time systems: Specification, Verification and analysis*. Prentice Hall International, 1996.
40. A. Burns and A. Wellings. Advanced fixed priority scheduling. In M. Joseph, editor, *Real-Time Systems: Specification, Verification and Analysis*, page 32–65. Prentice Hall, 1996.
41. R. W. Butler and G. B. Finelli. The infeasibility of experimental quantification of life-critical software reliability. *IEEE Transactions on Software Engineering*, 19(1):3–12, 1993.
42. C. c. Morgan and A. K. McIver. *pGCL:* Formal Reasoning for Random Algorithms. *South African Computer Journal*, 22, 1999. Appears in part at [181, Chap. 1].
43. S. Campos and E. Clarke. The Verus language: representing time efficiently with BDDs. In *AMAST Workshop on Real-Time-Systems, Concurrent and Distributed Software*, volume 1231 of *Lecture Notes in Computer Science*, page 64–78. Springer-Verlag, 1997.
44. S. Campos, E. Clarke, W. Marrero, M. Minea, and H. Hiraishi. Computing Quantitative Characteristics of Finite-State Real-Time Systems. In *IEEE Real-Time Systems Symposium*, 1994.
45. S. Campos, E. Clarke, and M. Minea. Symbolic Techniques for Formally Verifying Industrial System. *Science of Computer Programming*, 29(1–2):79–98, 1997.
46. F. Castor and P.H. M. Borba. A Language for Specifying Java Transformations. In *5th Brazilian Symposium on Programming Languages*, page 236–251, 2001.
47. A. L. C. Cavalcanti, P. Clayton, and C. O'Halloran. Control Law Diagrams in *Circus*. In J. Fitzgerald, I. J. Hayes, and A. Tarlecki, editors, *FM 2005: Formal Methods*, volume 3582 of *Lecture Notes in Computer Science*, page 253 –268. Springer-Verlag, 2005.
48. A. L. C. Cavalcanti and D. A. Naumann. A Weakest Precondition Semantics for Refinement of Object-oriented Programs. *IEEE Transactions on Software Engineering*, 26(8):713–728, 2000.
49. A. L. C. Cavalcanti and D. A. Naumann. Forward simulation for data refinement of classes. In L. Eriksson and P. A. Lindsay, editors, *FME 2002: Formal Methods — Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, page 471–490. Springer-Verlag, 2002.
50. A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. An Inconsistency in Procedures, Parameters, and Substitution the Refinement Calculus. *Science of Computer Programming*, 33(1):87–96, 1999.
51. A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A Refinement Strategy for *Circus*. *Formal Aspects of Computing*, 15(2 - 3):146–181, 2003.
52. A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. Unifying Classes and Processes. *Software and System Modelling*, 4(3):277–296, 2005.

53. K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley Publishing Company, 1988.
54. H. Chen, D. Dean, and D. Wagner. Model Checking One Million Lines of C Code. In *11th Annual Symposium on Network and Distributed System Security*, page 171–185, 2004.
55. Y. Chen and Z. Liu. From Durational Specifications to TLA Designs of Timed Automata. In *6th International Conference on Formal Engineering Methods*, volume 3308 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.
56. Y. Chen and Z. Liu. Integrating Temporal Logics. In E. A. Boiten, J. Derrick, and G. Smith, editors, *4th International Conference on Integrated Formal Methods*, volume 2999 of *Lecture Notes in Computer Science*, page 402–420. Springer-Verlag, 2004.
57. Y. Choueka. Theories of automata on ω-tapes: a simplified approach. *Journal of Computing Systems*, 8:117–141, 1974.
58. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *International Conference on Computer-Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
59. A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new Symbolic Model Verifier. In N. Halbwachs and D. Peled, editors, *11th Conference on Computer-Aided Verification*, number 1633 in Lecture Notes in Computer Science, page 495–499. Springer-Verlag, 1999.
60. E. Clarke and I. A. Draghicescu. Expressibility results for linear-time and branching-time logics. In *REX Workshop 1988*, volume 354 of *Lecture Notes in Computer Science*, page 428–437. Springer-Verlag, 1989.
61. E. M. Clarke, I. A. Draghicescu, and R. P. Kurshan. A unified approach for showing language containment and equivalence between various types of ω-automata. In A. Arnold and N. D. Jones, editors, *15th Colloquium on Trees in Algebra and Programming*, volume 431 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.
62. E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons for Branching Time Temporal Logic. In *Logics of Programs: Workshop*, volume 131 of *Lecture Notes in Computer Science*, page 52–71. Springer Verlag, 1981.
63. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programmings Languages and Systems*, 8(2):244–263, 1986.
64. E. M. Clarke, O. Grumberg, H. Hirashi, S. Jha, D. Long, and K. L. McMillan. Verification of the Future-Bus+ Cache Coherence Protocol. *Formal Methods in Systems Design*, 6(2):217–232, 1995.
65. E. M. Clarke, O. Grumberg, K. L. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *32nd ACM/IEEE Design Automation Conference*, 1995.
66. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
67. E. M. Clarke, M. Khaira, and X. Zhao. Word-Level Model Checking - Avoiding the Pentium FDIV error. In *33rd ACM/IEEE Design Automation Conference*, page 645–648, 1996.
68. E. M. Clarke, D. Kroening, J. Ouaknine, and O. Strichman. Computational challenges in bounded model checking. *International Journal on Software Tools for Technology Transfer*, 7(2), 2005.

69. Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-based Predicate Abstraction for ANSI-C. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *Lecture Notes in Computer Science*, page 570–574. Springer-Verlag, 2005.
70. J. Coenen and J. Hooman. Parameterized semantics for fault-tolerant real-time systems. In J. Vytopil, editor, *Formal Techniques in Real-Time and Fault Tolerant Systems*, page 51–78. Kluwer Academic Publishers, 1993.
71. J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, and Robby and H. Zheng. Bandera: extracting finite-state models from Java source code. In *International Conference on Software Engineering*, page 439–448, 2000.
72. M. L. Cornélio. *Object-Oriented Refactorings and Patterns as Formal Refinements*. PhD thesis, Centro de Informática, Universidade Federal de Pernambuco, 2004.
73. M. L. Cornélio, A. L. C. Cavalcanti, and A. C. A. Sampaio. Refactoring by Transformation. In *REFINE'2002*, volume 70 of *Eletronic Notes in Theoretical Computer Science*, page 641–660, 2002. Invited paper.
74. P. Cousot. *Handbook of theoretical computer science*, volume B, chapter Methods and logics for proving programs, page 841–993. Elsevier Science Publishers, j. van leeuwen edition, 1990.
75. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, page 238–252, 1977.
76. F. Cristian. A rigorous approach to fault-tolerant programming. *IEEE Transactions on Software Engineering*, SE-11(1):23–31, 1985.
77. B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1992.
78. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.
79. M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215, 1960.
80. D. Déharbe, S. Shankar, and E. Clarke. Model Checking VHDL with CV. In *Formal Methods in Circuit Automation Design*, number 1522 in Lecture Notes in Computer Science. Springer-Verlag, 1998.
81. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
82. D. Distefano. *On model-checking the dynamics of object-based software: a foundational approach*. PhD thesis, University of Twente, 2003.
83. S. Dunne. Recasting Hoare and He's Unifying Theories of Programs in the Context of General Correctness. In A. Butterfield and C. Pahl, editors, *5th Irish Workshop in Formal Methods*, BCS Electronic Workshops in Computing, 2001.
84. E. A. Emerson. *Handbook of theoretical computer science*, volume B, chapter Temporal and modal logic, page 995–1072. Elsevier Science Publishers, j. van leeuwen edition, 1990.
85. E. A. Emerson, A. K. Mok, A. P. Sistla, and J. Srinivasan. Quantitative Temporal Reasoning. In *2nd International Workshop on Computer Aided Verification*, number 531 in Lecture Notes in Computer Science, page 136–145. Springer-Verlag, 1990.
86. U. Engberg, P. Grønning, and L. Lamport. Mechanical Verification of Concurrent Systems with TLA. In G. Bochmann, editor, *4th International Conference on Computer Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, page 44–55. Springer-Verlag, 1992.

87. K. Etessami and G. J. Holzmann. Optimising Büchi automata. In *Concurrency Theory: 11th International Conference*, volume 1877 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.
88. J.-L. Fauquembergue, G. Kahn, W. Kubbat, S. Levedag, J.-L. Lions, L. Lübeck, L. Mazzini, D. Merle, and C. O'Halloran. Ariane 5 Flight 501 Failure Report. Technical report, Board of Inquiry, ESA, 1996.
89. Y. A. Feldman. A Decidable Propositional Dynamic Logic with Explicit Probabilities. *Information and Control*, 63:11–38, 1984.
90. Y. A. Feldman and D. Harel. A Probabilistic Dynamic Logic. *Journal of Computing and System Sciences*, 28:193–215, 1984.
91. C. J. Fidge and A. J. Wellings. An action-based formal model for concurrent, real-time systems. *Formal Aspects of Computing*, 9(2):175–207, 1997.
92. C. Fischer and H. Wehrheim. Model-Checking CSP-OZ Specifications with FDR. In K. Araki, A. Galloway, and K. Taguchi, editors, *1st International Conference on Integrated Formal Methods*, page 315–334. Springer-Verlag, 1999.
93. L. Fix and F. B. Schneider. Reason about programs by exploiting the environment. Technical Report TR94-1409, Department of Computer Science, Cornell University, 1994.
94. M. Fowler. *Refactoring*. Addison-Wesley, 1999.
95. J. Fröβl, J. Gerlach, and T. Kropf. An efficient algorithm for real-time model checking. In *European Design and Test Conference*, page 15–21, 1996.
96. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
97. P. H. B. Gardiner and C. C. Morgan. Data Refinement of Predicate Transformers. *Theoretical Computer Science*, 87:143 –162, 1991.
98. P. Gastin and D. Oddoux. Fast LTL to Büchi Automata Translation. In *Computer Aided Verification: 13th International Conference*, number 2102 in Lecture Notes in Computer Science. Springer-Verlag, 2000.
99. R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple On-The-Fly Automatic Verification of Linear Temporal Logic. In Chapman & Hall, editor, *IFIP/WG1.6 Symposium on Protocol Specification, Testing, and Verification*, page 3–18, 1995.
100. P. Gniesel and H. Koch. Static Composition of Refactorings. *Science of Computer Programming*, 52:9–51, 2004.
101. M. Goldsmith and P. Whittaker. A CSP front-end for probabilistic tools. Technical report, Deliverable D14, FORWARD Project, 2005.
102. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
103. S. Gottwald. Many-valued Logic. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Stanford University, 2004.
104. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *9th International Conference on Computer-Aided Verification*, number 1254 in Lecture Notes in Computer Science, page 72–83. Springer-Verlag, 1997.
105. G. Grimmett and D. Welsh. *Probability: an Introduction*. Oxford Science Publications, 1986.
106. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logics*. Foundations of Computing Series. MIT press, 2000.
107. S. Hart, M. Sharir, and A. Pnueli. Termination of Probabilistic Concurrent Programs. *ACM Transactions on Programming Languages and Systems*, 5:356–380, 1983.

108. J. Hatcliff, M. B. Dwyer, and Robby. Bogor: An Extensible Framework for Domain-Specific Model Checking. *Newsletter of European Association of Software Science and Technology*, 2004.
109. K. Havelund and T. Pressburger. Model Checking Java Programs Using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4), 2000.
110. T. Henzinger, Z. Manna, and A. Pnueli. Temporal proof methodologies for timed transition systems. *Information and Computation*, 112(2):273–337, 1994.
111. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software Verification with Blast. In *10th SPIN Workshop on Model Checking Software*, number 2648 in Lecture Notes in Computer Science, page 235–239. Springer-Verlag, 2003.
112. T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model checking for Real-Time Systems. *Information and Computation*, 111:193–244, 1994.
113. W. M. Ho, J. M. Jzquel, A. L. Guennec, and F. Pennaneac'h. UMLAUT: An Extendible UML Transformation Framework. In *14th IEEE International Conference on Automated Software Engineering*, 1999.
114. C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12:576–580, 1969.
115. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.
116. C. A. R. Hoare et al. Laws of Programming. *Communications of the ACM*, 30(8):672–686, 1987.
117. C. A. R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.
118. G. J. Holzmann. An Analysis of Bit-State Hashing. In Chapman & Hall, editor, *IFIP/WG1.6 Symposium on Protocol Specification, Testing, and Verification*, page 301–314, 1995.
119. G. J. Holzmann. State Compression in SPIN: Recursive Indexing and Compression Training Runs. In *Third SPIN Workshop*, 1997.
120. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
121. G. J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *2nd SPIN Workshop*, number 32 in DIMACS, page 23–32. Rutgers University, 1996.
122. G. J. Holzmann and M. H. Smith. Software Model Checking: Extracting verification models from source code. *Software Testing Verification and Reliability*, 11(2):65–79, 2001.
123. J. Hooman. *Specification and Compositional Verification of Real-Time Systems*, volume 558 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.
124. S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, 1990.
125. J. Hurd. *Formal Verification of Probabilistic Algorithms*. PhD thesis, Computing Laboratory, 2002.
126. T. Janowski and M. Joseph. Dynamic scheduling in the presence of faults: specification and verification. In B. Jonsson and J. Parrow, editors, *4th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1135 of *Lecture Notes in Computer Science*, page 279–298. Springer-Verlag, 1996.
127. T. Janowski and M. Joseph. Dynamic scheduling and fault-tolerance: Specification and verification. *Real-Time Systems*, 20(1):51–81, 2001.

128. He Jifeng and J. Bowen. Specification, Verification, and Prototyping of an Optimized Compiler. *Formal Aspects of Computing*, 6:643–658, 1994.
129. He Jifeng, Z. Liu, X. Li, and S. Qin. A relational model of object oriented programs. In *2nd Asian Symposium on Programming Languages and Systems*, volume 3302 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.
130. He Jifeng, Z. Liu, and S. Qin. A Relational Model for Object-Oriented Design. Technical report, UNU-IIST, 2005.
131. He Jifeng, K. Seidel, and A. K. McIver. Probabilistic Models for the Guarded Command Language. *Science of Computer Programming*, 28:171–192, 1997. Appears in part at [181, Chap. 5].
132. C. Jones. *Probabilistic Nondeterminism*. PhD thesis, Edinburgh University, 1990.
133. C. Jones and G. Plotkin. A Probabilistic Powerdomain of Evaluations. In *4th IEEE Symposium on Logic in Computer Science*, page 186–195. Computer Society Press, 1989.
134. C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, 1986.
135. M. Joseph and A. Goswami. What's 'Real' about Real-time Systems? In *IEEE Real-time Systems Symposium*, page 78–85. IEEE Computer Society Press, 1988.
136. M. Joseph and P. Pandya. Finding Response Times in a Real-time System. *Computer Journal*, 29(5):390–395, 1986.
137. R. Keller. Formal verification of parallel programs. *Communication of the ACM*, 19(7):371–384, 1976.
138. D. Knuth. Literate Programming. *Computer Journal*, 17, 1984.
139. R. Koymans. *Specifying message passing and Time-critical systems with temporal logic*. PhD thesis, Eindhoven University of Technology, 1989.
140. D. Kozen. A probabilistic PDL. *Journal of Computing and System Sciences*, 30(2):162–178, 1985.
141. R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes — The Automata-Theoretic Approach*. Princeton University Press, 1995.
142. L. Lamport. What good is temporal logic. In R. W. Mason, editor, *IFIP 9th World Congress*, page 657–668. North-Holland, 1983.
143. L. Lamport. Hybrid systems in TLA$^+$. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, page 77–102. Springer-Verlag, 1993.
144. L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.
145. L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Pearson Education, Inc., 2002.
146. L. Lamport and S. Merz. Specifying and Verifying Fault-Tolerant Systems. In H. Langmaak, W. P. de Roever, and J. Vytopil, editors, *3rd International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, page 41–76. Springer-Verlag, 1994.
147. L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problems. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.
148. K. G. Larsen and A. Skou. Bisimulation through Probabilistic Testing. *Information and Computation*, 94(1):1–28, 1991.
149. D. Lehmann and M. O. Rabin. On the Advantages of Free Choice: a Symmetric and Fully-Distributed Solutionto the Dining Philosophers Problem. In A. W. Roscoe, editor, *A Classical Mind: Essays in Honour of C. A. R. Hoare*, page 333–352. Prentice-Hall, 1984.

150. J. Lehoczky, L. Sha, and Y. Ding. The Rate-monotonic Scheduling Algorithms: Exact characterisation and average case behaviour. In *10th IEEE Real-time Systems Symposium*, page 261–270. IEEE Computer Society Press, 1989.
151. K. R. M. Leino. Recursive Object Types in a Logic of Object-oriented Programming. In C. Hankin, editor, *7th European Symposium on Programming*, volume 1381 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
152. N. Leveson. *Safeware*. Addison-Wesley, 1995.
153. N. Leveson, M. Heimdahl, H. Hildreth, and J. Reese. Requirements specification for Process-Control Systems. *IEEE Transactions on Software Engineering*, 20(9), 1994.
154. X. Li, Z. Liu, J. He, and Q. Long. Generating prototypes from a UML Model of requirements. In *International Conference on Distributed Computing and Internet Technology*, volume 3407 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.
155. J. Lilius and I. P. Paltor. vUML: A Tool for Verifying UML Models. In *14th IEEE International Conference on Automated Software Engineering*, page 255–258, 1999.
156. B. O. Lira, A. L. C. Cavalcanti, and A C. A. Sampaio. Automation of a Normal Form Reduction Strategy for Object-oriented Programming. In *5th Brazilian Workshop on Formal Methods*, page 193–208, 2002.
157. B. Littlewood and L. Strigini. Validation of Ultrahigh dependability for software-based systems. *Communications of the ACM*, 36(11):69–80, 1993.
158. C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):40–61, 1973.
159. Z. Liu. *Fault-Tolerant Programming by Transformations*. PhD thesis, Department of Computer Science, University of Warwick, 1991.
160. Z. Liu, J. He, and X. Li. rCOS: Refinement of object-oriented and component systems. In *International Symposium on Formal Methods of Component and Object Systems*, volume 2937 of *Lecture Notes in Computer SCience*. Springer-Verlag, 2004.
161. Z. Liu, J. He, X. Li, and Y. Chen. A relational model for object-oriented requirement analysis in UML. In *International Conference on Formal Engineering Methods*, volume 2885 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.
162. Z. Liu and M. Joseph. Transformation of Programs for Fault-Tolerance. *Formal Aspects of Computing*, 4(5):442–469, 1992.
163. Z. Liu and M. Joseph. Specifying and Verifying Recovery in Asynchronous Communicating Systems. In J. Vytopil, editor, *Formal Techniques in Real-Time and Fault Tolerant Systems*, page 137–166. Kluwer Academic Publishers, 1993.
164. Z. Liu and M. Joseph. Stepwise Development of Fault-Tolerant Reactive Systems. In H. Langmaack, W. P. de Roever, and J. Vytopil, editors, *3rd International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, page 529–546. Springer-Verlag, 1994.
165. Z. Liu and M. Joseph. Verification of Fault-Tolerance and Real-Time. In *26th Annual International Symposium on Fault-Tolerant Computing*, page 220–229. IEEE Computer Society, 1996.
166. Z. Liu and M. Joseph. Formalizing Real-Time Scheduling as Program Refinement. In M. Bertran and T. Rus, editors, *Transformation-Based Reactive Systems Development*, volume 1231 of *Lecture Notes in Computer Science*, page 294–309. Springer-Verlag, 1997.

167. Z. Liu and M. Joseph. Specification and Verification of Fault-tolerance, Timing and Scheduling. *ACM Transactions on Languages and Systems*, 21(1):46–89, 1999.
168. Z. Liu and M. Joseph. Verification, Refinement and Scheduling of Real-Time Programs. *Theoretical Computer Science*, 253(1), 2001.
169. Z. Liu, M. Joseph, and T. Janowski. Verification of schedulability of real-time programs. *Formal Aspects of Computing*, 7(5):510–532, 1995.
170. Z. Liu, A. P. Ravn, and X. Li. Unifying Proof Methodologies of Duration calculus and Linear Temporal Logic. *Formal Aspects of Computing*, 16(2), 2004.
171. Q. Long, Z. Liu, He Jifeng, and X. Li. Consistent Code Generation from UML Models. In *Australia Conference on Software Engineering*. IEEE Computer Scienty Press, 2005.
172. G. Lowe. *Probabilities and Priorities in Timed CSP*. PhD thesis, Oxford University Computing Laboratory, 1993.
173. G. Lowe and H. Zedan. Refinement of complex systems: A case study. Technical Report PRG-TR-2-95, Oxford University Computing Laboratory, 1995.
174. I. Lynce and J. P. Marques-Silva. Building State-of-the-Art SAT Solvers. In *European Conference on Artificial Intelligence*, page 166–170, 2002.
175. G. S. Avrunin M. B. Dwyer and J. C. Corbett. Patterns in Property Specifications for Finite-state Verification. In *21st International Conference on Software Engineering*, page 411–420, 1999.
176. Z. Manna and A. Pnueli. The temporal framework for concurrent programs. In R. S. Boyer and J. S. Moore, editors, *The Correctness Problem in Computer Science*, page 215–274. Academic Press, 1981.
177. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1991.
178. J. P. Marques-Silva and K. A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.
179. A. K. McIver and C. C. Morgan. Probabilistic Predicate Transformers: part 2. Technical Report PRG-TR-5-96, Programming Research Group, 1996. Appears revised at [181, Chap. 8].
180. A. K. McIver and C. C. Morgan. Partial Correctness for Probabilistic Demonic Programs. Technical Report PRG-TR-35-97, Programming Research Group, 1997.
181. A. K. McIver and C. C. Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Technical Monographs in Computer Science. Springer-Verlag, 2004.
182. A. K. McIver, C. C. Morgan, and T. Son Hoang. Probabilistic Termination in B. In *Formal Specification and Development in Z and B*, volume 2651 of *Lecture Notes in Computer Science*, page 216–239. Springer-Verlag, 2003.
183. A. K. McIver, C. C. Morgan, and E. Troubitsyna. The Probabilistic Steam Boiler: a Case Study in Probabilistic Data Refinement. In *International Refinement Workshop*. Springer Verlag, 1998. Also appears at [181, Chap. 4].
184. K. McMillan. *Symbolic Model Checking: an approach to the state explosion problem*. PhD thesis, Carnegie Mellon University, 1992. Technical Report CMU-CS-92-131.
185. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
186. K. L. McMillan and J. Schwalbe. *Shared Memory Multi-Processing*, chapter Formal Verification of the Gigamax Cache Coherency Protocol. MIT Press, 1992.

187. A. Mikhajlova and E. Sekerinski. Class refinement and Interface refinement in Object-oriented Programs. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *Industrial Benefit of Formal Methods*, volume 1313 of *Lecture Notes in Computer Science*, page 82–101. Springer-Verlag, 1997.
188. A. R. G. Milner. *Calculus of communicating systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.
189. R. Milner. *Communication and concurrency*. Prentice Hall, 1989.
190. M. Mislove. Nondeterminism and probabilistic choice: Obeying the laws. In *CONCUR 2000*, page 350–364, 2000.
191. A. Moitra and M. Joseph. Cooperative recovery from faults in distributed programs. In R. W. Mason, editor, *IFIP 9th World Congress*, page 481–486. North-Holland, 1983.
192. C. C. Morgan. *Programming from Specifications*. Prentice-Hall, 2nd edition, 1994.
193. C. C. Morgan. Proof Rules for Probabilistic Loops. In He Jifeng, J. Cooke, and P. Wallis, editors, *BCS-FACS 7th Refinement Workshop*, Workshops in Computing. Springer Verlag, 1996. `http://www.springer.co.uk/ewic/workshops/7RW`.
194. C. C. Morgan. The Generalised Substitution Language Extended to Probabilistic Programs. In *2nd International B Conference*, volume 1393 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
195. C. C. Morgan. Of probabilistic *wp* and CSP — and compositionality. In A. E. Abdallah, C. B. Jones, and J. W. Jones, editors, *25 Years and CSP*. Springer-Verlag, 2005.
196. C. C. Morgan and A. K. McIver. Abstraction and refinement in probabilistic systems. *ACM SIGMETRICS Performance Evaluation Review*, 32(4):41–47, 2005.
197. C. C. Morgan, A. K. McIver, and K. Seidel. Probabilistic Predicate Transformers. *ACM Transactions on Programming Languages and Systems*, 18(3):325–353, 1996. Also appears at [181, Chap. 5].
198. C. C. Morgan, A. K. McIver, K. Seidel, and J. W. Sanders. Refinement-Oriented Probability for CSP. *Formal Aspects of Computing*, 8(6):617–647, 1996.
199. J. M. Morris. A Theoretical Basis for Stepwise Refinement and the Programming Calculus. *Science of Computer Programming*, 9(3):287–306, 1987.
200. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. In *Design Automation Conference*, page 530–535, 2001.
201. R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.
202. J. von Neumann. Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components. In C. E. Shannon and J. Macarthy, editors, *In Automata Studies*, page 43–98. Princeton University Press, 1956.
203. T. Nipkow. Hoare Logics in Isabelle/HOL. In H. Schwichtenberg and R. Steinbrüggen, editors, *Proof and System-Reliability*, page 341–367. Kluwer, 2002.
204. J. Nordahl. *Specification and Design of Dependable Communicating Systems*. PhD thesis, Department of Computer Science, Technical University of Denmark, 1992.
205. P. Nuemann. *Computer Related Risks*. addison-Wesley, 1995.
206. C. O'Halloran, R. Arthan, and D. King. Using a formal specification contractually. *Formal Aspects of Computing*, 9(4), 1997.
207. C. O'Halloran, C. T. Sennett, and A. Smith. Specification of the compliance notation for SPARK and Z, (3 volumes). Technical Report DRA/CIS/CSE3/TR/94/27/1.2, DRA Malvern, 1994.

208. W. Opdyke. *Refactoring Object-oriented Frameworks*. PhD thesis, University of Illinois at Urban Champaign, 1992.
209. J. P. Kes P. Garbett, M. Shackleton, and S. Anderson. Secure synthesis of code: a process improvement experiment. In J. M. Wing, J. C. P. Woodcock, and J. Davies, editors, *FM'99: World Congress on Formal Methods*, volume 1709 of *Lecture Notes in Computer Science*, page 1816–1835. Springer-Verlag, 1999.
210. D. Parker, G. Norman, and M. Kwiatkowska. PRISM 2.0 Users' Guide. Technical Report DRA/CIS/CSE3/TR/94/27/1.2, GraphPad Software, 2004.
211. D. Peled. Combining Partial-Order Reduction with On-The-Fly Model Checking. In *6th International Conference on Computer-Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, page 377–390. Springer-Verlag, 1994.
212. M. Pilling, A. Burns, and K. Raymond. Formal specification and proofs of inheritance protocols for real-time scheduling. *Software Engineering Journal*, 5(5), 1990.
213. G. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, AArhus University, Computer Science Department, 1981.
214. A. Pnueli. The temporal logic of programs. In *18th IEEE Symposium on Foundations in Computer Science*, page 46–57, 1977.
215. A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Current Trends in Concurrency*, volume 224 of *Lecture Notes in Computer Science*, page 510–584. Springer-Verlag, 1986.
216. A. Pnueli and E. Harel. Applications of temporal logic to the specification of real-time systems. In M. Joseph, editor, *1st International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 331 of *Lecture Notes in Computer Science*, page 84–98. Springer-Verlag, 1988.
217. S. Qin, J. S. Dong, and W. N. Chin. A Semantic Foundation for TCOZ in Unifying Theories of Programming. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, page 321–340. Springer-Verlag, 2003.
218. J.-P. Queille and J. Sifakis. Specification and Verification of Concurrent Systems in CESAR: an Example. Technical report, IMAG-Universite Joseph Fourier, 1981. 254.
219. M. O. Rabin. The Choice-Coordination Problem. *Acta Informatica*, 17(2):121–134, 1982.
220. R. Ramos, A. C. A. Sampaio, and A. Mota. A Semantics for UML-RT Active Classes via Mapping into *Circus*. In *IFIP Conference on Formal Methods for Open Object-based Distributed Systems*, Lecture Notes in Computer Science, page 1–16. Springer-Verlag, 2005.
221. B. Randell. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, 1975.
222. B. Randell, P. A. Lee, and P. C. Treleaven. Reliability Issues in Computing Systems Design. *Computing Survey*, 10(2):123–165, 1978.
223. A. P. Ravn, H. Rischel, and K. M. Hansen. Specifying and verifying requirements of real-time systems. *IEEE Transactions on Software Engineering*, 19(1):41–55, 1993.
224. D. Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urban Champaign, 1999.
225. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall Series in Computer Science. Prentice-Hall, 1998.

226. A. W. Roscoe and C. A. R. Hoare. The Laws of occam Programming. *Theoretical Computer Science*, 60(2):177–229, 1988.
227. J. Ruf and T. Kropf. Symbolic model checking for a discrete clocked temporal logic with intervals. In *Advances in Hardware Design and Verification –International Conference on Correct Hardware and Verification Methods*, page 146–163, 1997.
228. A. C. A. Sampaio. *An Algebraic Approach to Compiler Design*, volume 4 of *AMAST Series in Computing*. World Scientific, 1997.
229. A. C. A. Sampaio, A. Mota, and R. Ramos. Class and Capsule Refinement in UML for Real Time. In *Workshop on Formal Methods*, volume 95 of *Electronic Notes in Theoretical Computer Science*, page 23–51, 2004.
230. A. C. A. Sampaio, J. C. P. Woodcock, and A. L. C. Cavalcanti. Refinement in *Circus*. In L. Eriksson and P. A. Lindsay, editors, *Formal Methods — Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, page 451–470. Springer-Verlag, 2002.
231. H. Schepers and R. Gerth. A compositional proof theory for fault-tolerant real-time systems. In *12th Symposium on Reliable Distributed Systems*, page 34–43. IEEE Computer Society Press, 1993.
232. R. D. Schlichting and F. B. Schneider. Fail-stop processors: an approach to designing fault tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, 1983.
233. D. A. Schmidt. *Denotational Semantics. A Methodology for Language Development*. Allyn and Bacon, Inc, 1986.
234. K. Seidel. *Probabilistic Communicating Processes*. PhD thesis, Oxford University, 1992.
235. K. Seidel, C. C. Morgan, and A. K. McIver. An Introduction to Probabilistic Predicate Transformers. Technical Report PRG-TR-6-96, Programming Research Group, 1996. Also appears in part at [181, Chap. 1].
236. C. T. Sennet. Demonstrating the Compliance of Ada Programs with Z Specifications. In C. B. Jones, R. C. Shaw, and T. Denvir, editors, *5th Refinement Workshop*, Workshops in Computing, page 70–87. Prentice-Hall, 1992.
237. S. Seres, J. Michael Spivey, and C. A. R. Hoare. Algebra of Logic Programming. In *International Conference on Logic Programming*, page 184–199, 1999.
238. M. Sharir, A. Pnueli, and S. Hart. Verification of Probabilistic Programs. *Science of Computer Programming*, 13(2):292–314, 1984.
239. W. Shen, K. Compton, and J. Huggins. A toolset for supporting UML static and dynamic model checking. In *Computer Software and Applications Conference*, page 147–152, 2002.
240. A. Sherif and He Jifeng. Towards a Time Model for *Circus*. In *International Conference in Formal Engineering Methods*, page 613–624, 2002.
241. U. Shrotri, P. Bhaduri, and R. Venkatesh. Model checking visual specification of requirements. In *International Conference on Software Engineering and Formal Methods*, page 202–209. IEEE Computer Society Press, 2003.
242. L. Silva, A. C. A. Sampaio, and E. Barros. A Constructive Approach to Hardware/Software Partitioning. *Formal Methods in System Design, Kluwer*, 24:45–90, 2004.
243. F. Somenzi. CUDD: CU Decision Diagram Package, 1998. University of Colorado at Boulder.
244. F. Somenzi and R. Bloem. Efficient Büchi Automata from LTL Formulae. In *12th International Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, page 248–263. Springer-Verlag, 2000.

245. R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.
246. A. Tarski. A Lattice Theoretical Fixed Point Theorem and its Applications. *Pacific Journal of Mathematics*, 5, 1955.
247. R. Tix, K. Keimel, and G. Plotkin. Semantic domains for combining probability and non-determinism. *Electronic Notes in Theoretical Computer Science*, 129:1–104, 2005.
248. R. J. van Glabbeek, S. A. Smolka, B. Steffen, and C. Tofts. Reactive, Generative and Stratified Models of Probabilistic Processes. In *IEEE Symposium on Logic in Computer Science*, 1990.
249. M. Y. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *1st IEEE Symposium on Logic in Computer Science*, page 322–331, 1986.
250. H. Wang. *Reflections on Kurt Gödel*. MIT Press, 2nd edition, 1988.
251. R. A. Weaver and T. P. Kelly. The Goal Structuring Notation - A Safety Argument Notation. In *Dependable Systems and Networks 2004 Workshop on Assurance Cases*, 2004.
252. E. V. Whiting and M. G. Hill. Safety Analysis of Hawk in Flight Monitor. In *Workshop on Program Analysis for Software Tools and Engineerin*, 1999.
253. B. A. Wichmann, A. Canning, D. Clutterbuck, L. Winsborrow, N. Ward, and D.W.R. Marsh. Industrial perspective on static analysis. *Software Engineering Journal*, 1995.
254. G. Winskel and M. Nielsen. *Handbook of Logic in Computer Science. Vol. 4: Semantic Modelling*, chapter Models for Concurrency, page 1–148. Oxford Science Publications, 1995.
255. J. C. P. Woodcock and A. L. C. Cavalcanti. The Semantics of *Circus*. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, page 184–203. Springer-Verlag, 2002.
256. J. C. P. Woodcock and A. L. C. Cavalcanti. A Tutorial Introduction to Designs in Unifying Theories of Programming. In E. A. Boiten, J. Derrick, and G. Smith, editors, *Integrated Formal Methods*, volume 2999 of *Lecture Notes in Computer Science*, page 40–66. Springer-Verlag, 2004. Invited tutorial.
257. J. C. P. Woodcock and J. Davies. *Using Z — Specification, Refinement, and Proof*. Prentice-Hall, 1996.
258. J. Yang, Q. Long, Z. Liu, and X. Li. A Predicative Semantic Model for Integrating UML Models. In *1st International Colloquium on Theoretical Aspects of Computing*, volume 3407 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.
259. L. Zhang and S. Malik. The Quest for Efficient Boolean Satisfiability Solvers. In *Computer Aided Verification: 14th International Conference*, page 17–36, 2002.
260. Y. Zhang and C. C. Zhou. A formal Proof of the Deadline Driven Scheduler. In H. Langmaak, W. P. de Roever, and J. Vytopil, editors, *3rd International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, page 756–775. Springer-Verlag, 1994.
261. C. C. Zhou, M. R. Hansen, A. P. Ravn, and H. Rischel. Duration Specifications for Shared Processors. In J. Vytopil, editor, *2nd International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems*, volume 571 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.
262. C. C. Zhou, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.

Author Index