Lecture Notes in Computer Science 3639

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

Patrice Godefroid (Ed.)

Model Checking Software

12th International SPIN Workshop
San Francisco, CA, USA, August 22-24, 2005
Proceedings

Volume Editor

Patrice Godefroid
Bell Laboratories, Lucent Technologies
2701 Lucent Lane, Lisle, IL 60532, USA
E-mail: god@bell-labs.com

Library of Congress Control Number: 2005930636

CR Subject Classification (1998): F.3, D.2.4, D.3.1, D.2

ISSN 0302-9743
ISBN-10 3-540-28195-9 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-28195-5 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11537328 06/3142 5 4 3 2 1 0

Preface

This volume contains the proceedings of the 12th International SPIN Workshop on Model Checking of Software, held in San Francisco, USA, on August 22–24, 2005. SPIN 2005 is a forum for practitioners and researchers interested in model-checking based techniques for the validation and analysis of communication protocols and software systems. The workshop focuses on topics including theoretical and algorithmic foundations and tools for software model checking, model derivation from code and code derivation from models, techniques for dealing with large and infinite state spaces, and applications. The workshop aims to foster interactions and exchanges of ideas with all related areas in software engineering. It has traditionally drawn contributions from both academia and industry.

The SPIN workshop series started 10 years ago, in 1995. Since then, SPIN workshops have been held on an annual basis at Montréal (1995), New Brunswick (1996), Enschede (1997), Paris (1998), Trento (1999), Toulouse (1999), Stanford (2000), Toronto (2001), Grenoble (2002), Portland (2003) and Barcelona (2004). All but the first SPIN workshop were organized as satellite events of larger conferences, in particular of CAV (1996), TACAS (1997), FORTE/PSTV (1998), FLOC (1999), the World Congress on Formal Methods (1999), FMOODS (2000), ICSE (2001, 2003) and ETAPS (2002, 2004). This year, SPIN was held as a satellite event of CONCUR 2005. The co-location of SPIN workshops with conferences has proven to be very successful and has helped to disseminate SPIN model checking technology to wider audiences. Since 1999, the proceedings of the SPIN workshops have appeared in Springer's Lecture Notes in Computer Science series.

The history of successful SPIN workshops is evidence for the maturing of software model-checking technology. While in earlier years the focus of the workshop series was algorithms and tool development around the SPIN model-checker, its scope was widened several years ago to include other software model-checking techniques, tools and applications.

This year, we received 45 regular paper submissions out of which 15 were selected. In addition, the SPIN 2005 program contained 4 tool presentation papers selected from 6 submissions. All submissions went through a rigorous reviewing process, where each paper received a mimimum of 3 referee reviews.

In addition to the refereed papers, three invited talks were given, by David Wagner (UC Berkeley) on *Pushdown Model Checking for Security*, Dawson Engler (Stanford University) on *Static Analysis Versus Model Checking for Bug Finding*, and Rajeev Alur (University of Pennsylvania) on *The Benefits of Exposing Calls and Returns* (invited talk of CONCUR/2005, and joint with SPIN/2005). Dawson Engler also contributed an original paper entitled *Execution Generated Test Cases: How to Make Systems Code Crash Itself* (co-authored

with Cristian Cadar), which can be found in these proceedings. The program also included three invited tutorials, by Gerard J. Holzmann (NASA JPL) and Theo C. Ruys (University of Twente) on *Effective Bug Hunting with* SPIN *and* MODEX, by Thomas A. Henzinger (EPFL), Ranjit Jhala (UCSD) and Rupak Majumdar (UCLA) on *The BLAST Software Verification System*, and by Willem Visser (NASA Ames) on *Model Checking Programs with Java PathFinder*.

I would like to thank the Program Committee members, as well as all the external reviewers who assisted them in their work. My thanks also go to the Steering Committee members and last year's organizers, Susanne Graf and Laurent Mounier, for their helpful advice. I would also like to thank the invited speakers and invited tutorial speakers, the authors of submitted papers, and the workshop participants. Special thanks go to Springer for providing us with the possibility of using a Conference Online Service free of charge and to the METAFrame team for their support. Last but not least, I would like to thank the organizers of CONCUR 2005, in particular Luca de Alfaro, for inviting us to hold SPIN 2005 as a satellite event and for their support and flexibility in accommodating the particular needs of the SPIN workshop.

June 2005 Patrice Godefroid

Organization

Program Committee

George Avrunin (U. Mass. Amherst, USA)
Dennis Dams (Bell Labs, USA)
Stefan Edelkamp (U. Dortmund, Germany)
Cormac Flanagan (UC Santa Cruz, USA)
Jaco Geldenhuys (Tampere U., Finland)
Patrice Godefroid (Bell Labs, USA; Chair)
Susanne Graf (Verimag, France)
Gerard Holzmann (NASA JPL, USA)
Sarfraz Khurshid (UT Austin, USA)
Stefan Leue (U. Konstanz, Germany)
Rupak Majumdar (UCLA, USA)
Laurent Mounier (Verimag, France)
Shaz Qadeer (Microsoft, USA)
Theo Ruys (U. Twente, Netherlands)
Willem Visser (NASA Ames, USA)
Pierre Wolper (U. Liège, Belgium)

Advisory Committee

Gerard Holzmann (NASA JPL, USA; Chair)
Amir Pnueli (Weizmann Inst., Israel)

Steering Committee

Thomas Ball (Microsoft, USA)
Susanne Graf (Verimag, France)
Stefan Leue (U. Konstanz, Germany)
Moshe Vardi (Rice U., USA)
Pierre Wolper (U. Liège, Belgium; Chair)

Additional Reviewers

Husain Aljazzar
Nina Amla
Dragan Boshnachki
Marius Bozga
Richard Chang
Pieter de Villiers
Paul Grisham
Henri Hansen
Klaus Havelund
Shahid Jabbar
Ranjit Jhala
Rajeev Joshi
Alberto Lluch-Lafuente
Oded Maler
Tilman Mehler

Kedar Namjoshi
Iulian Ober
Corina Pasareanu
Michael Perin
Nicolas Rouquette
Hassen Saïdi
Danhua Shao
Natalia Sidorova
Evghenia Stegantova
Brink Van der Merwe
Wei Wei

Table of Contents

Invited Talks/Papers

Invited Tutorials

State Representation and Abstraction

Dealing with Concurrency

Dealing with Complex Data

Checking Temporal Properties

Checking Security and Real-Time Properties

Tool Papers

Pushdown Model Checking for Security

David Wagner

U.C. Berkeley
daw@cs.berkeley.edu

Abstract. One of the key challenges for computer security is the problem of software security: how to build software that is free of implementation vulnerabilities. In this talk, I will present experience with pushdown model checking for software security. First, I will survey simple methods for pushdown model checking, and I will introduce MOPS, a tool for pushdown model checking of C programs. Then, I will show many security properties of interest may be encoded as temporal safety properties that are well-suited to analysis with model checking. I will report on our experience applying MOPS to tens of millions of lines of C code. Finally, I will discuss some possible directions for future research.

P. Godefroid (Ed.): SPIN 2005, LNCS 3639, p. 1, 2005.

Execution Generated Test Cases: How to Make Systems Code Crash Itself

Cristian Cadar and Dawson Engler*

Computer Systems Laboratory,
Stanford University,
Stanford, CA 94305, U.S.A

Abstract. This paper presents a technique that uses code to automatically generate its own test cases at run-time by using a combination of symbolic and concrete (i.e., regular) execution. The input values to a program (or software component) provide the standard interface of any testing framework with the program it is testing, and generating input values that will explore all the "interesting" behavior in the tested program remains an important open problem in software testing research. Our approach works by turning the problem on its head: we lazily generate, from within the program itself, the input values to the program (and values derived from input values) as needed. We applied the technique to real code and found numerous corner-case errors ranging from simple memory overflows and infinite loops to subtle issues in the interpretation of language standards.

1 Introduction

Systems code is difficult to test comprehensively. Externally, systems interfaces tend towards the baroque, with many different possible behaviors based on tricky combinations of inputs. Internally, their implementations tend towards heavily entangling nests of conditionals that are difficult to enumerate, much less exhaust with test cases. Both features conspire to make comprehensive, manual testing an enormous undertaking, so enormous that empirically, many systems code test suites consist only of a handful of simple cases or, perhaps even more commonly, none at all.

Random testing can augment manual testing to some degree. A good example is the fuzz [3, 4] tool, which automatically generates random inputs, which is enough to find errors in many applications. Random testing has the charm that it requires no manual work, other than interfacing the generator to the tested code. However, random test generation by itself has several severe drawbacks. First, blind generation of values means that it misses errors triggered by narrow ranges of inputs. A trivial example: if a function only has an error if its 32-bit integer argument is equal to "12345678" then random will most likely have to generate billions of test cases before it hits this specific case. Second, and

* This paper is a shortened version of [1], which was in simultaneous submission with similar but independent work by Patrice Godefroid et al [2]. Our thanks to Patrice for graciously accepting this version as an invited paper.

P. Godefroid (Ed.): SPIN 2005, LNCS 3639, pp. 2–23, 2005.

similarly, random testing has difficulty hitting errors that depend on several different inputs being within specific (even wide) ranges of values. Third, the ability of random testing to effectively generate random noise is also its curse. It is very poor at generating input that has structure, and as a result will miss errors that require some amount of correct structure in input before they can be hit. A clear example would be using random test generation to find bugs in a language parser. It will find cases where the parser cannot handle garbage inputs. However, because of the extreme improbability of random generation constructing inputs that look anything like legal programs it will miss almost all errors cases where the parser mishandles them.

Of course, random can be augmented with some amount of guidance to more intelligently generate inputs, though this comes at the cost of manual intervention. A typical example would be writing a tool to take a manually-written language grammar and use it to randomly generate legal and illegal programs that are fed to the tested program. Another would be having a specification or model of what a function's external behavior is and generate test cases using this model to try to hit "interesting" combinations. However, all such hybrid approaches require manual labor and, more importantly, a willingness of implementors to provide this labor at all. The reluctance of systems builders to write specifications, grammars, models of what their code does, or even assertions is well known. As a result, very few real systems have used such approaches.

This paper's first contribution is the observation that code can be used to *automatically* generate its own potentially highly complex test cases. At a high level, the basic idea is simple. Rather than running the code on manually-constructed concrete input, we instead run it on symbolic input that is initially allowed to be "anything." As the code observes this input, these observations tell us what legal values (or ranges of values) the input could be. Each time the code makes a decision based on an observation we conceptually fork the execution, adding on one branch the constraint that the input satisfies the observation, and on the other that it does not. We can then generate test cases by solving these constraints for concrete values. We call such tests *execution generated testing* (EGT).

This process is most easily seen by example. Consider the following contrived routine `bad_abs` that incorrectly implements absolute value:

```
0:   int bad_abs(int x) {
1:       if(x < 0)
2:           return -x;
3:       if(x == 12345678)
4:           return -x;
5:       return x;
6:   }
```

As mentioned before, even such a simple error will probably take billions of random-generated test cases to hit. In contrast, finding it with execution generated testing it is straightforward. Symbolic execution would proceed as follows:

1. Initial state: set `x` to the symbolic value of "anything." In this case, before any observations at all, it can be any value between `INT_MIN` and `INT_MAX`. Thus we have the constraints $x \geq INT\_MIN \wedge x \leq INT\_MAX$.
2. Begin running the code.
3. At the first conditional (line 1) fork the execution, setting `x` to the symbolic constraint $x < 0$ on the true path, and to $x \geq 0$ on the false path.
4. At the return (line 2) solve the constraints on `x` for a concrete value (such as `x == -1`). This value is later used used as a test input to `bad_abs`.
5. At the second conditional (line 3) fork the execution, setting `x` to the constraints $x \equiv 12345678 \wedge x \geq 0$ on the true path and $x \neq 12345678 \wedge x \geq 0$ on the false path.
6. At the second return (line 4) solve the symbolic constraints $x \equiv 12345678 \wedge x \geq 0$. The value is 12345678 is our second test case.
7. Finally, at line 5, solve x's constraints for a concrete value (e.g., `x = 1`). This value is used as our third, final case.

We can then test the code on the three generated values for `x`. Of course, this sketch leaves many open questions — when to generate concrete values, how to handle system calls, how to tell what is correct, etc. The rest of the paper discusses these issues in more detail.

There are a couple of ways to look at the approach. From one point of view, implementation code has a "grammar" of the legal inputs it accepts and acts on, or rejects. EGT is an automatic method to extract this grammar (and the concrete sentences it accepts and rejects) from the implementation rather than from a hand-written specification. From another viewpoint, it can be seen as a way to turn code "inside out" so that instead of consuming inputs becomes a generator of them. Finally, and perhaps only half-vacuously, it can be viewed as a crude analogue of the Heisenberg effect in the sense that unlike observations perturbing experiments from a set of potential states into a variety of concrete ones, observations in this case perturb a set of possible inputs into a set of increasingly concrete ones. The more precise the observation the more definitively it perturbs the input. The most precise observation, an equality comparison, fixes the input to a specific concrete value. The least precise, an inequality, simply disallows a single value but leaves all others as possibilities.

This paper has three main contributions:

1. A simple conceptual approach to automatically generate test cases by running code on symbolic inputs.
2. A working prototype EGT system.
3. Experimental results showing that the approach is effective on real code.

The paper is organized as follows. Section 2 gives an overview of the method. Section 3 discusses concrete implementation issues. The next four sections give four case studies of applying the approach to systems code. Finally, Section 7 discusses related work and Section 8 concludes.

2 Overview

This section gives an overview of EGT. The next section discusses some of the implementation details.

In order to generate test cases, EGT runs the code on symbolic rather than real input. Whenever code reads from its environment (via network packets, command line options, files, etc) we want to instead return a symbolic variable that has no constraints on its actual value. As the program executes and uses or observes this value (e.g., through comparisons), we add constraints based on these observations. Then, to determine how to reach a given program path, we solve these constraints and generate input that satisfies them.

At a high-level, the EGT system has three core activities:

1. Instrumentation to track symbolic constraints. Our prototype EGT system instruments the tested code using a source-to-source transformation. This instrumentation inserts checks around every assignment, expression and branch in the tested program and calls into our runtime system. It also inserts code to fork a new process at each decision point at which the associated boolean condition could return both `true` and `false`.
2. Constraint solving. We model our constraints using formulas of quantifier-free first-order logics as represented by CVCL, a state-of-the-art decision procedure solver [5, 6]. CVCL has been used in applications ranging hardware verification to program analysis to mathematical theorem proving.
   We use CVCL in two ways. First, after every branch point we call it to determine if the current set of constraints is satisfiable. If not, we stop following the code path, otherwise we continue. CVCL is sound: if it states that no solution exists, it is correct. Second, at the end of a code path that uses symbolic input, we use CVCL to generate concrete values to use as test input.
3. Modeling. External functions that return or consume input can either be modeled so that they work with symbolic variables, or not modeled, in which case any value they take must be made concrete. In general, one can leave most things unmodeled, with the downside that testing coverage will be reduced. Models are not that hard to write. A four-line model for the Unix `recv` system call is given in Section 6. In addition, models can be used to speed up the test generation. This optimization is discussed in Section 3.2.

The mechanical act of instrumenting code is pretty easy, and there are a lot of constraint solvers to pick from and use as black boxes. Thus, the main challenge for the approach is how to run code symbolically. The next subsection talks about this in more detail.

2.1 Symbolic Execution

The basic idea behind our approach is that when we perform logical or arithmetic operations, we generate constraints for these, and when we perform control flow

decisions, we fork execution and go down both paths. This section sketches how we can symbolically execute code. For ease of exposition, we initially assume that all the variables in a program are symbolic; Section 3.1 shows how we can intermix symbolic and concrete execution in order to efficiently process real code.

Assignment: `v = e` . We symbolically do an assignment of an expression `e` to a variable `v` by generating the constraint that $v \equiv e$. For example, `v = x + y` generates the constraint that $v \equiv x + y$; other arithmetic and logical operators are similar.

The complication is that `v` may have been involved in previous constraints. We must distinguish the newly assigned value of `v` from its use in any already generated constraints. For example, assume we have two assignments: (1) `x = y` and then (2) `y = 3`. The first assignment will generate the constraint that $x \equiv y$. The second will generate the constraint $y \equiv 3$. At this point, the constraints imply $x \equiv 3$, which is obviously nonsensical. This new value for `y` after its assignment `y = 3` has nothing to do with any prior constraints involving `y` and should have no impact on them. Thus, an assignment `v = e` must have two parts. First, generate a new location for `v` and only then generate the constraint that $v \equiv y$. [1]

If-statements. We symbolically execute an if-statement as follows: (1) fork execution at the conditional, (2) on the true path add the constraint that the conditional expression `e` is true ($e \equiv true$) and continue, (3) on the false path add the constraint that `e` is false ($e \equiv false$) and continue. For example:

```
 concrete |      symbolic
if(e)     | if(fork() == child)
  s1;     |   add_constraint(e == true);
          |   s1;
else      | else
  s2;     |   add_constraint(e == false);
          |   s2;
```

Loops. We transform loops into if-statements with goto's so they are handled as above. One danger is that iterating on a symbolic loop variable can continue forever, forking a new execution on each evaluation of the loop condition. The usual practical hack is to only iterate a fixed number of times or for a fixed amount of time (we do the latter). Neither solution is perfect. However, in our context almost any solution is preferable to manual test generation.

Function calls: `f(x)`. There are three differences between a symbolic function call and an imperative, call-by-value call. First, control can return multiple times into the caller, once for each fork-branching that occurs. Second, constraints placed on `x` in the body of `f` must propagate up to the caller. For example, the concrete code:

[1] Alternatively, ignoring aliasing, we could have equivalently gone through all existing constraints involving `v` and relabeled them to use a new, fresh name.

```
int foo(int x) {
    if(x == 3)
        return 1;
    else
        return 2;
}
```

will generate a symbolic execution that returns twice into the caller, since the branch will cause a forked execution. On the true branch we want to propagate the constraint that $x \equiv 3$ back to the caller and on the false that $x \neq 3$. The final difference is that at the exit point from a function, we create a temporary symbolic variable and return that as the function's expression value. Figure 1 gives a symbolic translation of `bad_abs` based on the above rules.

```
// initial constraints: x >= INT_MIN /\ x <= INT_MAX
int symbolic_bad_abs(int x) {
    ret = new symbol;  // holds the return expression.

    if(fork() == child) // fork execution at each branch point.
        add_constraint(x < 0); add_constraint(ret = -x);
        // first return, final constraints:
        //     x >= INT_MIN /\ x <= INT_MAX /\ x < 0 /\ ret = -x
        return ret;
    else
       add_constraint(x >= 0);

    if(fork() == child) // fork execution
        add_constraint(x = 12345678); add_constraint(ret = -x);
        // second return, final constraints: x >= INT_MIN /\ x <= INT_MAX
        //                 /\ x >= 0 /\ x = 12345678 /\ ret = -x
        return ret;
    else
        add_constraint(x != 12345678);

     add_constraint(ret = x);
     // last return final constraints: x >= INT_MIN /\ x <= INT_MAX
     //                     /\ x >= 0 /\ x != 12345678 /\ ret = x
     return ret;
}
```

Fig. 1. A symbolic translation of `bad_abs`

2.2 What is Correctness?

EGT, like all testing approaches, needs to have some notion of what "bad" behavior is so that it can flag it. We use three approaches to do so.

First, and unsurprisingly, check for program independent properties, such as segmentation faults, storage leaks, memory overflows, division by zero, deadlocks, uses of freed memory, etc.

Second, do cross-checking. If a piece of code implements an important interface, then there are likely to be several implementations of it. These implementations can be cross-checked against each other by running the test cases generated from one implementation (or both) on both implementations and flagging differences. One important usage model: after modifying a new version of a system, cross-check it against the old version to make sure any change was intended. This approach works especially well for complex interfaces.

Third, specification-by-example. While writing specifications to state what exactly code must do in general is hard, it is often much easier to take the specific test cases our tool generates and specify what the right answers are just for these cases. For example, for the `bad_abs` routine, the EGT system generates the three concrete values: -3, 12345677, 12345678. Thus, for testing we would just do:

```
assert(bad_abs(−3) == 3);
assert(bad_abs(12345677) == 12345677);
assert(bad_abs(12345678) == 12345678);
```

3 Implementation Issues

This section discusses implementation aspects of our EGT tool.

3.1 Mixed Symbolic and Concrete Execution

Ignoring memory and solver-limitations, we can run any code entirely symbolically until it interacts with the outside, concrete world. For example, if it calls external code, or sends a packet on a real network to a machine running concrete code, or prints output to be read by a real person. At this point you must either make the inputs to the external code concrete (e.g, you must send data rather than a symbolic constraint in a network packet), or, alternatively, make a model of the world to pull it into the simulation.

In practice, constraint solvers are not as robust as one might hope and so without care overzealous constraint generation will blow them up, sometimes for good theoretic reasons, sometimes for unimpressive practical ones. Further, symbolic-only execution is expensive in both speed and space. Thus, we do a hybrid approach that intermixes concrete and symbolic execution. The basic approach is that before every operation we dynamically check if the values are all concrete. If so, we do the operation concretely. Otherwise, if at least one value is symbolic we do the operation symbolically (using the logic described in Section 2.1.

We use the CIL tool [7] to instrument the code of tested programs. Below, we sketch how to conceptually rewrite source constructs for a C-like language so that they can run on either concrete or symbolic values, mentioning some of the more important practical details.

Our first transformation conceptually changes each variable or expression `v` to have two instances: a concrete one (denoted `v.concrete`) and a symbolic one (denoted `v.symbolic`). If `v` is concrete, `v.concrete` holds its concrete value and `v.symbolic` contains the special token $\langle invalid \rangle$. Conversely, if `v` is symbolic, `v.symbolic` holds its symbolic value and `v.concrete` is set to $\langle invalid \rangle$.

In practice, we track the `v.symbolic` field using a table lookup that takes the address of a the variable `v` (which gives it a unique name) and returns `v`'s associated "shadow" symbolic variable `v.symbolic` (if it is symbolic) or null (if it is concrete). In the latter case, the variable `v` contains the concrete value (`v.concrete`) and can just be used directly. The following examples assume explicit concrete and symbolic fields for clarity.

```
assign_rule(T &v, T e) {
  if(e is concrete)
    // equivalent to v.concrete = e.concrete;
    //               v.symbolic = <invalid>;
    v = (concrete=e.concrete, symbolic=<invalid>);
  else
    // equivalent: v.symbolic = e.symbolic
    v = (concrete=<invalid>, symbolic=new symbolic var T);
    constraint(v.symbolic = e.symbolic);
}
```

Fig. 2. Rewrite rule for assignment `v = e` for any variable `v` and expression `e` of type `T`

The most basic operation is assignment. Figure 2 gives the basic assignment rule. If the right hand variable `e` is a concrete expression or variable, just assign its concrete value to the left-hand side `v` and mark `v`'s symbolic component as `invalid`. If `e` is symbolic, then as explained in the previous section, we must allocate a fresh symbolic variable to be used in any new constraints that are generated. After that, we first set `v.concrete` to be `invalid` and then add the constraint that `v.symbolic` equals `e.symbolic`.

Roughly as simple are basic binary arithmetic operators. Figure 3 gives the rewrite rule for binary addition; other binary arithmetic operators are similar. If both `x` and `y` are concrete, we just return an expression whose concrete part is just their addition and symbolic part is invalid. Otherwise we build a symbolic constraint `s` and then return an expression that has `s` as its symbolic component and `invalid` for its concrete.

The rewrite rule for if-statements is a straight-forward combination of the purely symbolic rule for if-statements with the similar type of concrete-symbolic checking that occurs in binary relations. There are two practical issues. First, our current system will happily loop on symbolic values — the parent process of a child doing such looping will terminate it after a timeout period expires. Second, we use the Unix `fork` system call to clone the execution at every symbolic

```
// rule for x + y
T plus_rule(T x, T y) {
    if(x and y are concrete)
       return (concrete=x.concrete + y.concrete, <invalid>);

    s = new symbolic var T;
    if(x is concrete)
        constraint(s = x.concrete + y.symbolic);
    else if y is concrete
        constraint(s = x.symbolic + y.concrete);
    else
        constraint(s = x.symbolic + y.symbolic);
    return (concrete=<invalid>, symbolic=s);
}
```

Fig. 3. Rewrite rule for "`x + y`" where variables `x` and `y` are of type `T`

branch point. Naively this will quickly lead to an exponential number of processes executing. Instead we have the parent process wait for the child to finish before continuing to execute on its branch of the conditional. This means we essentially do depth-first search where there will only be one active process and a chain of its predecessors who are sleeping waiting for the active process to complete.

```
// rule for *p
T deref_rule(T* p) {
   if(*p is concrete)
       return (concrete=*p, symbolic=<invalid>);
   else
       s = new symbolic var T;
       if(p is concrete)
          constraint(s = (*p).symbolic);
       else
          // symbolic dereference of p
          constraint(s = deref(p.symbolic));
       return (concrete=<invalid>, symbolic=s);
}
```

Fig. 4. Rewrite rule for dereference "`*p`" of any pointer `p` of type `T`. The main complication occurs when we dereference a symbolic pointer: in this case we must add a symbolic constraint on the dereferenced value.

Because dereference deals with storage locations, it is one of the least intuitive rewrite rules. Figure 4 gives the rewrite rule for dereferencing `*p`. A concrete dereference works as expected. A dereference of a concrete pointer `p` that points to a symbolic value also works as expected (i.e., just like assignment, except that the rvalue is dereferenced). However, if `p` itself is symbolic, then we cannot

actually dereference it to get what it points to but instead must generate a funny constraint that says that the result of doing so equals the symbolic dereference of `p`.

At an implementation level, CVCL currently does not handle symbolic dereferences so we do not either. Further, in the short term we do not really do the right thing with any pointer dereference that involves a symbolic value (such as a symbolic offset off of a concrete pointer or a symbolic index into a symbolic array). In such cases we will generate a concrete value, which may be illegal.

One happy result of this limitation is that, when combined with the the way the implementation uses a lookup table to map variables to their shadow symbolic values, it makes handling address-of trivial. For example, given the assignment `p = &v` we simply do the assignment, always, no matter if `v` is a symbolic or concrete. A lookup of `p` will return the same symbolic variable (if any) that lookup of `&v` does. Thus any constraints on it are implicitly shared by both. Alternatively, if there is no symbolic, then `p` will point directly at the concrete variable and dereference will work as we want with no help.

Function calls are rewritten similarly to the previous section.

One implementation detail is that to isolate the effects of the constraint solver we run it in its own child Unix process so that (1) we can kill it if it does not terminate and (2) any problems it runs into in terms of memory or exceptions are isolated.

3.2 Creating a Model for Speed

Not all the code in the program under testing should be given the same level of attention. For example, many of our benchmarks make intensive use of the string library, but we don't want to generate test cases that exercise the code in these string routines.

More precisely, imagine a program which uses `strcmp` to compare two of its symbolic strings. Most implementations of `strcmp` would traverse one of the strings, and would compare each character in the first string with the corresponding character in the second string and would return a value when the two characters differ or when the end of a string has been reached. Thus, the routine would return to the caller approximately 2n times, each time with a different set of constraints. However, most applications use a routine such as `strcmp` as a black box, which could return only one of the following three values: 0, when the strings are equal, -1 when the first string is lexicographically smaller than the second one, and 1 otherwise. Returning the same value multiple times does not make any difference for the caller of the black box.

Instead of instrumenting routines such as those in the string library, we could instead provide models for them. A model for `strcmp` would return three times, once for each possible return value. After each fork, the model would add a series of constraints which would make the outcome of that branch symbolically true: for example, on the branch which returns 0, the model would add constraints setting the two strings equal. Of course, certain branches may be invalid; e.g. if

the two string have different lengths, `strcmp` could not return 0. In this case, the corresponding branch is simply terminated.

We implemented models for the routines in the string library, and used them in generating tests for our benchmarks. Adding these specifications has two main benefits. On the one hand, it removes useless test cases from the generated test suites (by removing tests which would only improve code coverage in the string routines), and on the other hand it significantly improves performance. For the WsMp3 benchmark that we evaluate in Section 6, the test suites are generated approximately seven times faster.

3.3 Discussion

Currently we do lazy evaluation of constraints, deferring solving them until the last possible moment. We could instead do eager evaluation, where as soon as we use a symbolic value we make up a concrete one. This eliminates the need to execute code symbolically. However, by committing to a concrete value immediately, it precludes the ability to change it later, which will often be necessary to execute both paths of any subsequent branch based on that variable's value (since the concrete value will either satisfy the true or the false branch, but not both). A hybrid approach might be best, where we make up concrete values immediately and then only do full symbolic execution on code paths that this misses.

4 Micro-case Study: Mutt's UTF8 Routine

As the first micro-benchmark to evaluate EGT, we applied it to a routine used by the popular Mutt email client to convert strings from the UTF-8 to the UTF-7 format. As reported by Securiteam, this routine in Mutt versions up to version 1.4 have a buffer overflow vulnerability which may allow a malicious IMAP server to execute arbitrary commands on the client machine [8].

We selected this paper in part because it has been one of the examples in a recent reliability paper [9], which used a carefully hand-crafted input to exploit it.

We extracted the UTF8 to UTF7 conversion routine from Mutt version 1.4, ran the code through our tool, and generated test cases for different lengths of the UTF-8 input string. Running these generated tests immediately found the error.

The paper we took the code from suggested a fix of increasing the memory allocation ratio from `n*2` to `n*7/3`. We applied this change to the code, and reran the EGT generated test cases, which immediately flagged that the code still has an overflow. The fact that the adjusted ratio was still incorrect highlights the need for (and lack of) automated, comprehensive testing.

Table 1 presents our results. For each input size, we report the size of the generated test suite and the time it took to generate it, the cumulative statement coverage achieved up to and including that test suite, and the largest output size

that we generated for that input size. These results (and all our later results), were generated on a Intel Pentium 4 Mobile CPU at 1.60GHz, with 512MB RAM.

Table 1. Test suites generated for utf8_to_utf7

Input Size	Generation Time	Test Suite Size	Statement Coverage	Largest Output
1	16s	10	84.0%	5
2	1m35s	38	94.2%	8
3	7m26s	132	94.2%	11
4	34m12s	458	95.6%	15
5	2h35m	1569	95.6%	19

5 Case Study: Printf

This section applies EGT to three different `printf` implementations. The `printf` routine is a good example of real systems code: a highly complex, tricky interface that necessitates an implementation with thickets of corner cases. Its main source of complexity is the output format string it takes as its first argument. The semantics of this single string absorb the bulk of the 234 lines the ANSI C99 standard devotes to defining `printf`; these semantics define an exceptionally ugly and startling programming language (which even manages to include iteration!).

Thus, `printf` is a best-case scenario for EGT. The standard and code complexity create many opportunities for bugs. Yet the inputs to test this complexity can be readily derived from `printf`'s parsing code, which devolves to fairly simple, easily solved equality checks. Further, the importance of `printf` means there are many different implementations, which we can use to finesse the need for a specification by cross-checking against each other.

We checked the following three `printf` implementations; all of them (intentionally) implemented only a subset of the ANSI C99 standard:

1. The Pintos instructional operating systems `printf`; the implementation intentionally elides floating point. This implementation is a stern test of EGT, since the developer (the co-author of a widely-read C book) had intimate knowledge of the standard.
2. The `gccfast printf`, which implements a version of printf in terms of `fprintf`. [2]
3. A reduced-functionality `printf` implementation for embedded devices. [3]

We used EGT to generate test suites by making the format string the single symbolic argument to `printf`. We set the size of this symbolic string to a fixed

[2] `http://www.opensource.apple.com/darwinsource/WWDC2004/gccfast-1614/`
[3] `http://www.menie.org/georges/embedded/index.html`

Table 2. Test suites generated for `printf`, the first row of each size gives the number of generated tests, the second row the time required to do so

Format Length	Pintos´ printf	Embedded printf	GCCfast printf
2	34 21s	17 2s	30 15s
3	356 4m0s	75 1m48s	273 3m10s
4	3234 40m47s	337 21m6s	2105 87m36s
128	590 123m56s	72 119m38s	908 120m19s

Table 3. Mismatches found in the printf implementations

	Pintos´ printf	Embedded printf	GCCfast printf
Mismatches self tests	426 of 4214	146 of 501	7 of 3316
Mismatches all tests	624 of 8031	6395 of 8031	91 of 8031
Statement Coverage	95% (172 lines)	95% (101 lines)	98% (63 lines)

length and generated test cases from the resultant constraints. We describe our measurements below and then discuss the bugs and differences found.

Measurements. We generated test cases for format strings of length 2, 3, 4, and 128. Table 2 shows the test suite size that we generated for each format length and the time it took to generate the test suite. We allowed a maximum of 30 seconds per CVCL query; there were only two queries killed after spending more than 30 seconds. For format lengths of 128 long, we terminated the test generation after approximately two hours.

Below are a representative fraction of EGT-generated format strings of length 4:

```
" %lle" " %#0f" " %G%." " % +l" " %#he" " %00." " %+jf"
" %-lf" " %#hf" " %+f " %#.E" " %00 " " %.c " " %
" % #c" " %-#." " %c%'" " %c%j" " %# p" " %---" " %+-u"
" %llc" " %0g " " %#+-" " %0 u" " %9s%"
```

Note that while almost all look fairly bizarre, because they are synthesized from actual comparisons in the code, many are legal (and at some level "expected" by the code).

Results. After generating test suites, we checked the output for each `printf` in two ways. First, we took the tests each implementation generated and cross-checked its output on these tests against the output of `glibc`'s `printf`. Each

of of the three implementations attempts to implement a subset of the ANSI C99 standard, while `glibc` intends to fully implement it. Thus, any difference is a potential bug. EGT discovered lots of such differences automatically: 426 in Pintos, 146 in the Embedded `printf` and 7 in `GCCfast`'s `printf` (which was surprising since it only does minimal parsing and then just calls `fprintf`, which then calls `glibc`'s `printf`). Since we had access to the implementor of Pintos we focused on these; we discuss these below.

Second, we took the tests generated by all implementations and cross-checked their output against each other. Since they intentionally implement different subsets of the standard, we expect them to have different behavior. This experiment tests whether EGT can find such differences automatically. It can: 624 in Pintos, 6395 in `Embedded` and 91 in `GCCfast`.

Note that in both experiments, the Pintos and the GCCfast `printf` routines print an error message and abort when they receive a format string that they cannot handle. Since they only intend to handle a subset of the standard, this is correct behavior, and we do not report a mismatch in this case. In contrast, the `Embedded printf` instead fails silently when it receives a format string which it cannot handle. This means that we cannot differentiate between an incorrect output of a handled case and an unhandled case, and thus we report all these cases as mismatches.

Table 3 also shows the statement coverage achieved by these test suites; all printf's achieve more than 95% coverage. Most of the lines that were not covered are unreachable. For example, Pintos' `printf` has a `NOT_REACHED` statement which should never be reached as long as Pintos treats all possible format strings. Similarly, for the Embedded `printf`, we don't reach the lines which redirect the output to a string buffer instead of `stdout`; these lines are used by `sprintf`, and never by `printf`. Some lines however where not reached because our system treats only the format string as symbolic, while the rest of the arguments are concrete. Finally, two of the three `printf` implementations use non-standard implementations for determining whether a character is a digit, which our system does currently not handle correctly. The number of lines reported in Table 3 are real lines of code, that is lines which have at least one instruction.

We reported all mismatches from Pintos to its developer, Ben Pfaff. We got confirmation and fixes of the following bugs.

Incorrect grouping of integers into groups of thousands.

> "Dammit. I thought I fixed that... Its quite obviously incorrect in that case." — Ben Pfaff, unsolicited exclamation, 3/23/05, 3:11pm.

The code mishandled the "'" specifier that says to comma-separate integer digits into groups of three. The exact test case was:

```
// correct: -155,209,728
// pintos : -15,5209,728
printf("%'d", -155209728);
```

Amusingly enough, the bug had been fixed in the developer's tree, but he had forgotten to push this out to the released version (which we were testing).

Incorrect handling of the space and plus flags.

> "That case is so obscure I never would have thought of it." — Ben Pfaff, unsolicited exclamation, 3/23/05, 3:09pm.

The character "%" can be followed by a space flag, which means that "a blank should be left before a positive number (or empty string) produced by a signed conversion" (`man printf(3)`). Pinto incorrectly leaves a blank before an unsigned conversion too. We found a similar bug for the plus flag.

This bug and the previous error both occurred in the same routine, `format_integer`, which deals with formating integers. The complexity of the specification of even this one small helper function is representative of the minutia-laden constraints placed on many systems interfaces and their internals.

We now give a more cursory description of the remaining errors.

Incorrect alignment of strings. Pintos incorrectly handles width fields with strings, although this feature works correctly for integers (which got better testing).

Incorrect handling of the t and z flags. When the flag t is used, the unsigned type corresponding to `ptrdiff_t` should be used. This is a detail of the standard which was overseen by the developer. We found a similar bug for the z flag, which specifies that the signed type corresponding to `size_t` should be used.

No support for wide strings and chars. Pintos does not support wide string and wide chars, but fails silently in this case with no error message.

Undefined behavior. We found several bugs which are caused by underspecified features. An example of such a case is "`printf(``%hi'', v)`, whose output is undefined if `v` cannot be represented as a `short`.

6 Case Study: WsMp3

This section applies our technique to the WsMp3 web server designed for transferring MP3 files [10]. We use WsMp3 version 0.0.5 which, uninstrumented contains about 2,000 lines of C code; instrumented about 40,000. This version contains a security vulnerability that allows attackers to execute arbitrary commands on the host machine [11, 12]. Our technique automatically generated test cases that found this security hole. In addition, it found three other memory overflows and an infinite loop caused by bad network input (which could be used for a DoS attack).

We first discuss how we set up test generation, coverage results, and then the most direct method of effectiveness: bugs found.

6.1 Setting Up WsMp3

WsMp3 has the typical web server core: a main loop that listens for connections using `accept`, reads packet from the connection using `recv`, and then does

operations based on the packet value. It also has a reasonably rich interaction with the operating system. As a first cut we only made the network packet's returned by `recv` be symbolic, but made the packet size be concrete. We did so by replacing calls to `recv` with calls to a model of it (`recv_model`) that just "returned" a symbolic array of bytes of a specific length:

```
// [model does not generate failures; msg_len is fixed]
ssize_t recv_model(int s, char *buf, size_t len, int flags) {
  make_bytes_symbolic(buf, msg_len);
  return msg_len;
}
```

It "reads in" a message of length `msg_len` by telling the system the address range between `buf` and `buf+msg_len` should be treated as symbolic. We then generated test cases for one byte packet, two bytes, and so forth by changing `msg_len` to the desired length.

After the web server finishes processing a message, we inserted a call into the system to emit concrete values associated with the message's constraints. We then emit these into a test file and run the web server on it.

One subtlety is that after the web server processes a single message we exit it. Recall that at every conditional on a symbolic value (roughly) we fork execution. Thus, the web server will actually create many different children, one for each branch point. Thus, even processing a "single" message will generate many many test messages. In the context of this server, one message has little to do explicitly with another and thus we would not get any more test cases by doing additional ones. However, for a more stateful server, we could of course do more than one message.

Finally, it was not entirely unheard of for even the symbolic input to cause the code to crash during test generation. We handle segmentation faults by installing a handler for the `SIGSEGV` signal and, if it is invoked, generate a concrete test case for the current constraints and then exit the process.

Since WsMp3 makes intensive use of the standard string library, we used our own `string.h` library described in Section 3.2. In our tests, using this library improves performance by roughly seven-fold.

6.2 Test Generation Measurements

We used EGT testing to generate tests for packets of size 1, 2, 3, 4, 5, 12, and 128. Table 4 gives (1) the number of tests generated for each size, (2) the time it took (user time), and (3) the number of times the CVCL constraint solver failed to generate a concrete test from a set of constraints within 30 seconds.

Given our naive implementation, the test generation time was non-trivial. For packets of size 12 and 128 we stopped it after 14 hours (they were running on a laptop that we wanted to write this paper on). However, note that in some sense high test generation cost is actually not so important. First, test generation happens infrequently. The frequent case, running the generated tests, takes less than a minute. Second, test generation is automatic. The time to manually

Table 4. Test suites generated for WsMp3. We stopped test generation for size 12 and 128 after roughly 14 hours

Packet Size	Unfinished Queries	Execution Time (s)	Test Suite Size
1	0	0s	1
2	0	0s	1
3	0	57s	18
4	0	10m28s	90
5	8	16m13s	97
12	134	14h15m	1173
128	63	14h15m	165

generate tests that would get similar amounts types of path coverage would be enormous. Further, manual generation easily misses cases silently. Finally, as far as we know, there was no test suite for WsMp3. Clearly the EGT alternative is much better.

We compare coverage from EGT to random testing. We use statement coverage generated using `gcc` and `gcov`. We would have preferred a more insightful metric than line coverage, but were not able to find adequate tools. We generated random tests by modifying the `recv` routine to request messages filled with random data of a given size. For each packet size (1, 2, 3, 4, 5, 128, 256, and 512 bytes long), we generate 10, 1000, and 100,000 random tests, and then measured the cumulative statement coverage achieved by all these tests. We recorded a statement coverage of 23.4%, as opposed to 31.2% for EGT.

However, the roughly 8% more lines of code hit by EGT is almost certainly a dramatic underreporting of the number of distinct paths it hits. More importantly, these lines appear out of reach of random testing no matter how many more random tests we do. In addition, note that it takes about two hours and a half to execute all the random test cases, while it takes less than a minute to execute all the EGT test cases.

We manually examined the code to see why EGT missed the other statements. Many of the lines of code that were not hit consisted of debugging and logging code (which was disabled during testing), error reporting code (such as printing an error message and aborting when a call to malloc fails), and code for processing the command-line arguments (which wasn't all reached because we didn't treat the arguments as symbolic inputs).

However, a very large portion of the code was not reached because the request messages that we fabricate do not refer to valid files on the disk, or because we fail to capture several timing constraints. As an example from the first category, when a `GET` request is received, the web server extracts the file name from the request packet, and then it checks if the file exists by using `fopen`. If the file does not exist, WsMp3 sends a corresponding error message to the client. If the file is valid, the file name is passed through various procedures for further processing. Since we don't have any files on our server, and since almost all the files being fabricated by our system would be invalid anyway, the code which process files

and file names is never invoked. The right way to solve this problem is to provide models for functions such as `fopen`, `fread`, and `stat`. However, even without these models, we find interesting errors, as the next subsection describes.

6.3 Errors Found

We have identified five errors in the code which parses the request messages received by WsMp3. All were caused by a series of incorrect assumptions that WsMp3 makes about the request being processed. We describe three illustrative bugs below.

```
// [buf holds network message]
char* get_op(char *buf) {
     char* op;
     int i;

     if((op=(char *)malloc(10))==NULL) {
          printf("Not enough memory!\n");
          exit(1);
     }
     // [note: buf is '0' terminated]
    if(buf!=NULL && strlen(buf)>=3) {
          //strncpy(op,buf,3);
          i=0;
          while(buf[i]!=' ') {
             op[i]=buf[i];
             i++;
          }
          op[i]='\0';
    }
    else op=NULL;

    return op;
}
```

Fig. 5. WsMp3 buffer overflow bug: occurs if received message (held in `buf`) has more than 10 characters before the first space

Figure 5 gives the first bug. Here WsMp3 assumes that the first part of the request message (held in `buf`) holds the type of the client request, such as `GET` or `POST`, separated from the rest of the message by a space. After a request is received, WsMp3 copies this action type in an auxiliary buffer by copying all the characters from the original request, until a space is encountered. Unfortunately, it assumes the request is legal rather than potentially malicious and allocates only ten bytes for this buffer. Thus, if it receives an invalid request which does not contain a space in the first ten characters, the buffer overflows and WsMp3

usually terminates with a segmentation fault. Amusingly, there is a (commented out) attempt to instead do some sort of copy using the safe `strncpy` routine which will only up to a pre-specified length.

This routine is involved in a second bug. As part of the checking it does do, it will return `NULL` if the input is `NULL` or if the size of the incoming message is less than three characters. However, the caller of this routine does not check for a `NULL` return and always passes the buffer to `strcmp`, causing a remote-triggered segmentation fault.

The third final bug was interesting: for certain rare request messages (where the sixth character is either a period or a slash, and is followed by zero or more periods or slashes, which are immediately followed by a zero), WsMp3 goes into an infinite loop. Our EGT system automatically generates the very unusual message required to hit this bug. The problematic code is shown below:

```
while (cp[0] == '.' || cp[0] == '/')
  for (i=1; cp[i] != '\0'; i++) {
      cp[i-1] = cp[i];
      if (cp[i+1] == '\0')
         cp[i] = '\0';
    }
```

7 Related Work

To the best of our knowledge, while there has been work related to test generation and synthesis of program inputs to reach a given program point, there is no previous approach that effectively generates comprehensive tests automatically from a real program. There certainly exists no tool that can handle systems code. We compare EGT to past test generation work and then to bug finding methods.

Static test and input generation. There has been a long stream of research that attempts to use static techniques to generate inputs that will cause execution to reach a specific program point or path.

One of the first papers to attack this problem, Boyer at al. [13], proposes the use of symbolic execution to follow a given path was in the context of a system, SELECT, intended to assist in debugging programs written in a subset of LISP. The usage model was that the programmer would *manually* mark each decision point in the path that they wanted executed and the system would incrementally attempt to satisfy each predicate. More recently, researchers have tended to use static analysis to extract constraints which then they try to solve using various methods. One example is Gotlieb et al [14], who statically extracted constraints which they tried to solve using (naturally) a constraint solver. More recently, Ball [15] statically extracted predicates (i.e., constraints) using "predicate abstraction" [16] and then used a model checker to try to solve these predicates for concrete values. There are many other similar static efforts. In general, static techniques are vastly weaker than dynamic at gathering the type of information needed to generate real test cases. They can deal with limited amounts of fairly

straightforward code that does not interact much (or at all) with the heap or complex expressions, but run into intractable problems fairly promptly.

Dynamic techniques test and input generation. Much of the test generation work relies on the use of a non-trivial manually-written specification of some kind. This specification is used to guide the generation of testing values ignoring the details of a given implementation. One of the most interesting examples of such an approach is Korat [17], which takes a specification of a data-structure (such as a linked list or binary tree) and exhaustively generates all non-isomorphic data structures up to a given size, with the intention of testing a program using them. They use several optimizations to prune data structure possibilities, such as ignoring any data structure field not read by a program. EGT differs from this work by attempting to avoid any manual specification and targeting a much broader class of tested code.

Past automatic input generation techniques appear to focus primarily on generating an input that will reach a given path, typically motivated by the (somewhat contrived) problem of answering programmer queries as to whether control can reach a statement or not. Ferguson and Korel[18] iteratively generate tests cases with the goal of hitting a specified statement. They start with an initial random guess, and then iteratively refine the guess to discover a path likely to hit the desired statement. Gupta et al. [19] use a combination of static analysis and generated test cases to hit a specified path. They define a loss function consisting of "predicate residuals" which roughly measures by "how much" the branch conditions for that path were not satisfied. By generating a series of test cases, they use a numerical solver to find test case values that can trigger the given path. Gupta's technique combines some symbolic reasoning with dynamic execution, mitigating some of the problems inherit in either approach but not in both. Unfortunately, the scalability of the technique has more recently been called into question, where small systems can require the method to take an unbounded amount of time to generate a test case [20].

In EGT differs from this work by focusing on the problem of comprehensively generating tests on all paths controlled by input. This prior work appears to be much more limited in this regard.

Software Model Checking. Model checkers have been previously used to find errors in both the design and the implementation of software systems [21–26, 22]. These approaches tend to require significant manual effort to build testing harnesses. However, to some degree the approaches are complementary: the tests our approach generates could be used to drive the model checked code.

Generic bug finding. There has been much recent work on bug finding [27, 26, 28, 29]. Roughly speaking because dynamic checking runs code, it is limited to just executed paths, but can more effectively check deeper properties implied by code. For example that the code will infinite loop on bad inputs, that a formatting command is not obeyed correctly. Many of the errors in this paper would be difficult to get statically. However, we view static analysis as complementary

to EGT testing — it is lightweight enough that there is no reason not to apply it and then use EGT.

8 Conclusion

This paper has proposed a simple method of automatically generating test cases by executing code on symbolic inputs called execution generated testing. We build a prototype EGT system and applied it to real code. We found numerous corner-case errors ranging from simple memory overflows and infinite loops to subtle issues in the interpretation of language standards.

These results, and our experience dealing with and building systems suggests that EGT will work well on systems code, with its often complex requirements and tangled logic.

Acknowledgements

The authors thank Ted Kremenek for his help with writing and related work and David Dill for writing comments. The authors especially thank Ben Pfaff for his extensive help with the code and results in Section 5. This research was supported by NSF ITR grant CCR-0326227, NSF CAREER award CNS-0238570-001, and a Junglee Corporation Stanford Graduate Fellowship.

References

1. Cadar, C., Engler, D.: Execution generated test cases: How to make systems code crash itself. Technical Report CSTR 2005-04 3, Stanford University (2005)
2. Godefroid, P., Klarlund, N., Sen, K.: Dart: Directed automated random testing. In: Proceedings of the Conference on Programming Language Design and Implementation (PLDI), Chicago, IL USA, ACM Press (2005)
3. Miller, B.P., Fredriksen, L., So, B.: An empirical study of the reliability of UNIX utilities. Communications of the Association for Computing Machinery **33** (1990) 32–44
4. Miller, B., Koski, D., Lee, C.P., Maganty, V., Murthy, R., Natarajan, A., Steidl, J.: Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical report, University of Wisconsin - Madison (1995)
5. Barrett, C., Berezin, S.: CVC Lite: A new implementation of the cooperating valid ity checker. In Alur, R., Peled, D.A., eds.: CAV. Lecture Notes in Computer Science, Springer (2004)
6. Ganesh, V., Berezin, S., Dill, D.L.: A decision procedure for fixed-width bit-vectors. Unpublished Manuscript (2005)
7. Necula, G.C., McPeak, S., Rahul, S., Weimer, W.: Cil: Intermediate language and tools for analysis and transformation of c programs. In: International Conference on Compiler Construction. (2002)
8. Securiteam: Mutt exploit. `http://www.securiteam.com/unixfocus/5FP0T0U9FU.html` (2003)
9. Rinard, M., Cadar, C., Dumitran, D., Roy, D.M., Leu, T., William S. Beebee, J.: Enhancing server availability and security through failure-oblivious computing. In: Symposium on Operating Systems Design and Implementation. (2004)

10. : Wsmp3 webpage. http://wsmp3.sourceforge.net/ (2005)
11. Associates, C.: Wsmp3 exploit. http://www3.ca.com/securityadvisor/vulninfo/Vuln.aspx?ID=15609 (2003)
12. Secunia: Wsmp3 exploit. http://secunia.com/product/801/ (2003)
13. Boyer, R.S., Elspas, B., Levitt, K.N.: Select – a formal system for testing and debugging programs by symbolic execution. ACM SIGPLAN Notices **10** (1975) 234–45
14. Gotlieb, A., Botella, B., Rueher, M.: Automatic test data generation using constraint solving techniques. In: ISSTA '98: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis, ACM Press (1998) 53–62
15. Ball, T.: A theory of predicate-complete test coverage and generation. In: FMCO'2004: Symp. on Formal Methods for Components and Objects, Springer-Press (2004)
16. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of c programs. In: PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation, ACM Press (2001) 203–213
17. Boyapati, C., Khurshid, S., Marinov, D.: Korat: Automated testing based on Java predicates. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA). (2002) 123–133
18. Ferguson, R., Korel, B.: The chaining approach for software test data generation. ACM Trans. Softw. Eng. Methodol. **5** (1996) 63–86
19. Gupta, N., Mathur, A.P., Soffa, M.L.: Automated test data generation using an iterative relaxation method. In: SIGSOFT '98/FSE-6: Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering, ACM Press (1998) 231–244
20. Edvardsson, J., Kamkar, M.: Analysis of the constraint solver in una based test data generation. In: ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering, ACM Press (2001) 237–245
21. Holzmann, G.J.: The model checker SPIN. Software Engineering **23** (1997) 279–295
22. Godefroid, P.: Model Checking for Programming Languages using VeriSoft. In: Proceedings of the 24th ACM Symposium on Principles of Programming Languages. (1997)
23. Holzmann, G.J.: From code to models. In: Proc. 2nd Int. Conf. on Applications of Concurrency to System Design, Newcastle upon Tyne, U.K. (2001) 3–10
24. Brat, G., Havelund, K., Park, S., Visser, W.: Model checking programs. In: IEEE International Conference on Automated Software Engineering (ASE). (2000)
25. Corbett, J., Dwyer, M., Hatcliff, J., Laubach, S., Pasareanu, C., Robby, Zheng, H.: Bandera: Extracting finite-state models from java source code. In: ICSE 2000. (2000)
26. Ball, T., Rajamani, S.: Automatically validating temporal safety properties of interfaces. In: SPIN 2001 Workshop on Model Checking of Software. (2001)
27. Das, M., Lerner, S., Seigle, M.: Path-sensitive program verification in polynomial time. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, Berlin, Germany (2002)
28. Coverity: SWAT: the Coverity software analysis toolset. http://coverity.com (2005)
29. Bush, W., Pincus, J., Sielaff, D.: A static analyzer for finding dynamic programming errors. Software: Practice and Experience **30** (2000) 775–802

Effective Bug Hunting with Spin and Modex

Gerard J. Holzmann[1] and Theo C. Ruys[2]

[1] NASA/JPL, Laboratory for Reliable Software
http://spinroot.com/gerard/
[2] University of Twente, The Netherlands
http://www.cs.utwente.nl/~ruys/

Abstract. This tutorial consists of two parts. In the first part we present an advanced overview of Spin [1,4], and illustrate its practical application to logic model checking problems. In the second part of the tutorial we present an overview of a related tool called Modex [2,3]. Modex can be used to extract Spin verification models directly from C source code. It supports the definition of user-defined abstractions, and cleverly exploits the capability in Spin version 4 to include embedded C code inside abstract verification models. We will show how to use Spin and Modex, separately and combined, in an effective way when searching for design errors in distributed software applications. Both Spin and Modex are written in ANSI-C and can freely be used on research projects.

The first part of this tutorial is meant for intermediate to advanced Spin users. The objective is to illustrate the effective application of both Promela and Spin, giving solutions to frequently encountered verification problems and discussing some useful recipes for the use of logic model checkers. We will also take a look 'under the hood' to briefly describe the architecture of Spin's verification engine, and then show how one can exploit this information in building Promela verification models that include embedded C code constructs.

The second part of the tutorial shows how Modex can be used to extract verification models from C code. This process relies on a user-definition of an abstraction table to guide the model extraction process. We will show how this methodology was first used for the exhaustive verification of a commercial telephone switch, developed at Bell Labs between 1998 and 2001. The verification procedure based on model extraction and model checking for multi-threaded code proved to be significantly more effective in its bug finding capabilities than the standard software testing process.

References

1. G. J. Holzmann. *The* Spin *Model Checker – Primer and Reference Manual.* Addison-Wesley, Boston, Massachusetts, USA, 2004.
2. G. J. Holzmann and M. H. Smith. An Automated Verification Method for Distributed Systems Software Based on Model Extraction. *IEEE Transactions on Software Engineering*, 28(4):364–377, April 2002.
3. Modex Homepage. URL: http://cm.bell-labs.com/cm/cs/what/modex/.
4. Spin Homepage. URL: http://spinroot.com/spin/.

P. Godefroid (Ed.): SPIN 2005, LNCS 3639, p. 24, 2005.

The BLAST Software Verification System*

Thomas A. Henzinger[1], Ranjit Jhala[2], and Rupak Majumdar[3]

[1]EPFL
[2]UC San Diego
[3]UC Los Angeles

BLAST is a verification system for checking safety properties of C programs. BLAST implements a lazy-abstraction algorithm, which integrates automatic abstraction refinement and model checking [8]. The input to BLAST is a C program and a safety monitor written in a specification language with C like syntax [1]. The lazy-abstraction algorithm returns either an error trace of the program together with a corresponding test case [2], or a proof that the program satisfies the safety property [6] (or, since the problem is undecidable, the algorithm may fail to terminate). BLAST automatically constructs and refines a parsimonious predicate abstraction of the input program, using an interpolation-based decision procedure to find, based on counterexample analysis, the relevant predicates for each individual control location [5].

BLAST has successfully verified and found violations of interface safety properties of large device driver programs [6,5], memory safety properties [3], race conditions in nesC programs (using an extension for concurrent programs) [4], and file handling properties of large open-source programs [9]. Extensions to BLAST support program testing [2] and incremental programming [7].

BLAST is available from http://www.eecs.berkeley.edu/∼blast.

References

1. D. Beyer, A. Chlipala, T.A. Henzinger, R. Jhala, and R. Majumdar. The BLAST query language for software verification. In *SAS 04*, LNCS 3148, pp. 2–18. Springer, 2004.
2. D. Beyer, A. Chlipala, T.A. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *ICSE 04*, pp. 326–335. ACM, 2004.
3. D. Beyer, T.A. Henzinger, R. Jhala, and R. Majumdar. Checking memory safety with BLAST. In *FASE 05*, LNCS 3442, pp. 2–18. Springer, 2005.
4. T.A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *PLDI 04*, pp. 1–13. ACM, 2004.
5. T.A. Henzinger, R. Jhala, R. Majumdar, and K.L. McMillan. Abstractions from proofs. In *POPL 04*, pp. 232–244. ACM, 2004.
6. T.A. Henzinger, R. Jhala, R. Majumdar, G.C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *CAV 02*, LNCS 2404, pp. 526–538. Springer, 2002.

* This research was supported in part by the NSF grants CCR-0234690, CCR-0225610, ITR-0326577, and CCR-0427202.

P. Godefroid (Ed.): SPIN 2005, LNCS 3639, pp. 25–26, 2005.

7. T.A. Henzinger, R. Jhala, R. Majumdar, and M.A.A. Sanvido. Extreme model checking. In *Verification: Theory and Practice*, LNCS 2772, pp. 332–358. Springer, 2003.
8. T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL 02*, pp. 58–70. ACM, 2002.
9. R. Jhala and R. Majumdar. Path slicing. In *PLDI 05*. ACM, 2005.

Model Checking Programs with Java PathFinder

Willem Visser[1] and Peter Mehlitz[2]

[1] Research Institute for Advanced Computer Science (RIACS)
[2] Computer Sciences Corporation (CSC),
NASA Ames Research Center, Moffett Field, CA 94035, USA
{wvisser, pcmehlitz}@email.arc.nasa.gov

In recent years there has been an increasing move towards analyzing software programs with the aid of model checking. In this tutorial we will focus on one of the first model checkers developed specifically for analyzing programs - Java PathFinder (JPF). JPF was awarded the 2003 Engineering Innovation award from NASA's Office of Aerospace Technology. JPF is freely available and the development became an open-source project in April 2005[1]. JPF has been used on numerous NASA applications, including, Mars Rover control, Deep-Space 1 fault protection, and Shuttle ground control software as well as on software from companies such as Fujitsu.

JPF is an explicit-state model checker that analyzes Java programs on the bytecode level. Since it works on the bytecode level, it can deal with all Java's language features, including, concurrency, dynamic class loading, dynamic creation of threads and objects, garbage collection, exception handling, etc. The tutorial will highlight the main capabilities of the tool and also its current weaknesses. One of the core design decisions was to create a modular tool that could easily be understood and extended by others. A core component of the tutorial will be an introduction to the tool architecture as well as the features making it extensible (*Listener* interfaces and the *Model Java* interface). In addition we will discuss the features of the tool that make model checking Java programs tractable, these will include, state compression and storage, dynamic partial-order reduction and using search heuristics.

To give an indication of the current research direction of JPF the last part of the tutorial will focus on the tool's new features, such as the symbolic execution and test-case generation facilities. JPF supports symbolic execution of linear integer arithmetic as well dynamically allocated structured data (e.g. linked lists, red-black trees, etc.). We will show how a simple extension of JPF allows the combination of symbolic execution, predicate abstraction and shape analysis for efficient test-input generation.

We will conclude the tutorial with a discussion of our experiences of using the tool for the past five years and where we believe the biggest challenges for software model checking is in the future.

[1] http://javapathfinder.sourceforge.net

P. Godefroid (Ed.): SPIN 2005, LNCS 3639, p. 27, 2005.

An Incremental Heap Canonicalization Algorithm

Madanlal Musuvathi[1] and David L. Dill[2]

[1] Microsoft Research, Redmond
madanm@microsoft.com
[2] Computer Systems Laboratory, Stanford University
dill@cs.stanford.edu

Abstract. The most expensive operation in explicit state model checking is the hash computation required to store the explored states in a hash table. One way to reduce this computation is to compute the hash incrementally by only processing those portions of the state that are modified in a transition. This paper presents an *incremental* heap canonicalization algorithm that aids in such an incremental hash computation. Like existing heap canonicalization algorithms, the incremental algorithm reduces the state space explored by detecting heap symmetries. On the other hand, the algorithm ensures that for small changes in the heap the resulting canonical representations differ only by relatively small amounts. This reduces the amount of hash computation a model checker has to perform after every transition, resulting in significant speedup of state space exploration. This paper describes the algorithm and its implementation in two explicit state model checkers, CMC and Zing.

1 Introduction

There is a practical need to apply verification techniques to large software systems. In this vein, explicit state model checkers that systematically enumerate the possible states of a given system have been successful in finding complex errors, and sometimes proving their absence in software systems [1,2,3,4].

Apart from the well known state explosion problem, one challenge in scaling explicit state model checkers to large systems is the sheer size of individual system states. These model checkers store the explored states in a hash table to avoid exploring them redundantly. The hash computation required in this process is the most time consuming operation during model checking [5]. At the minimum, computing a hash value requires a few arithmetic operations for *every* byte of the state. This can be very expensive, especially for states that are tens or hundreds of kilobytes.

One way to reduce the hash computation overhead is to compute the hash *incrementally*. During model checking, a transition is very likely to modify only small portions of the state. By accounting for these differences between the initial and final state of a transition, and using a suitable hash function, a model checker can generate the hash value of the final state by incrementally updating

P. Godefroid (Ed.): SPIN 2005, LNCS 3639, pp. 28–42, 2005.

the hash value of the initial state. (See §2.3 for details.) This amortization of hash computation over the unmodified portion of the state can significantly improve the speed of state space exploration.

Implementing such an incremental hashing scheme is complicated by the need for software model checkers to perform *heap canonicalization* [6,7]. Almost all non-trivial programs allocate memory in the heap. Heap canonicalization is a state space reduction technique that enables model checkers to identify states that are behaviorally equivalent but differ only in the memory locations of heap objects. To identify such equivalent states, a model checker executes a heap canonicalization algorithm to transform a state to a canonical representation that is unique for all equivalent states. This canonical representation is then inserted in the hash table.

Unfortunately, the only known heap canonicalization algorithm [7] does not admit incremental processing. This algorithm performs a depth first traversal of the heap, and the canonical representation of each object depends on its depth first ordering. As a result, even small structural changes to the heap, such as object additions or deletions can modify the canonicalization of a large number of objects in the heap. Thus, even when a transition modifies small portions of the state, the canonical representations of the initial and final states can be significantly different. This forces the model checker to process large portions of the state during hash computation.

This paper presents an improved, *incremental* heap canonicalization algorithm. Like [7], this algorithm generates a unique canonical representation for all equivalent heap states. However, this algorithm ensures that small changes to the heap only result in relatively small changes in the canonical representation. This allows the model checker to perform incremental hash computation. The basic idea of the incremental algorithm is to determine the canonicalization of a heap object from its *shortest path* to some global variable. When a transition makes small changes to the heap structure, the shortest path of most objects is likely to remain the same [8,9]. A model checker only needs to process those objects whose shortest paths have changed in a transition.

We have implemented this incremental algorithm in two explicit state model checkers, CMC [4] and Zing [10]. The algorithm is very easy to implement, and can handle arbitrary data structures in the heap, including type-unsafe pointers. We have applied the algorithm for several large models and have achieved an improved performance in all of them (§5). For the examples we have tried the model checker processes at most 5% of the objects in the heap during hash computation using the incremental heap canonicalization algorithm. This results in a model checking speedup of 2 to 9 times. While providing this speedup, the incremental algorithm preserves the state space reduction provided by the previous heap canonicalization algorithm [7].

2 Preliminaries

This section describes the necessary formalisms required to explain the main ideas in the paper. Motivated by our efforts in checking C programs, we will

not assume that the heap pointers are *type-safe*. Specifically, we will allow a pointer variable to point to objects of different types at different instances in the program. Also, we will allow these variables to point to arbitrary fields *in* the object.

2.1 Heap Objects

At any instant, the program state includes a collection of heap objects occupying memory addresses from a countably infinite set H. To allow arbitrary pointer arithmetic, we define the normal arithmetic operations, such as $+, \leq$, over H in the obvious way.

A heap object is identified by its start address $\in H$. For an object p, $len(p)$ represents the length of the object. The fields of an object p of length l occupy memory locations in the range $[p, p+l)$. For simplicity, assume that a field of an object p is named by its integer offset f in p, where $0 \leq f < len(p)$. The term $p[f]$ denotes the value of a field f in p. For our purposes, we will assume that $p[f]$ is either an integer, a pointer value $\in H$, or the special *null* pointer. An object p *points to* another object q exactly when there is a field f of p such that $q \leq p[f] < q + len(q)$. In this case, $p[f]$ is a *pointer* to q and p is said to *contain* a pointer to q.

2.2 Program State

Apart from the heap, the state of a program at any instant consists of all the global variables and the stacks of all the threads. We seek to capture the entire state of the program in the heap as follows.

Conceptually, all the global variables in a program can be considered as fields of a global *root* object that represents the statically allocated region in memory. For ease of exposition, assume that the addresses of all global variables are in H. Also, the stack of each thread is represented as a linked-list of stack frames, where each stack frame is a heap object. We assume that for each thread there is a field in *root* that points to its stack. [1]

The program state S is a set of objects $\{root = p_0, p_1, p_2, \ldots, p_n\}$. Given a state S, we assume that all objects in S do not overlap in memory. In other words, for any $p_i, p_j \in S, p_i \neq p_j \Rightarrow p_i < p_j \vee p_i \geq p_j + len(p_j)$. Also, we assume that all objects in a state are *reachable* from the *root* object. That is, for any object $p \in S$, there is a sequence $\langle root = p_0, p_1, \ldots, p_n = p \rangle$ such that p_i points to p_{i+1} for $0 \leq i < n$. We assume that any object that is not reachable from the *root* object is automatically removed from the state. [2]

[1] We also assume that each thread is *statically* identified by a unique identifier. Detecting *thread-symmetries* by permuting these identifiers is an interesting problem, but beyond the scope of this paper.

[2] Such memory-leak detection (or garbage collection) can be done during the canonicalization algorithm described in Section 4.

2.3 Incremental Hashing of State

The performance bottleneck in an explicit model checker is the hash computation required when storing the state in a hash table. One way to mitigate the performance overhead is to compute the hash *incrementally* as follows.

For a given state, the model checker computes a *partial* hash value for each object in the state. The hash value of the entire state is obtained from these partial hash values. The goal is to cache these partial hash values with objects, and recompute them only when a transition explored by the model checker modifies the object. However, if an object remains unmodified in a transition, its cached value can be used when computing the hash value of the final state. This amortization of hash computation across states improves the model checking performance.

Formally, we assume the existence of three hash functions h_o, h_S, and h_u defined as follows. For an object p of length l, there is a hash function h_o that takes $l + 1$ arguments. The partial hash value of p is given by

$$\mathcal{H}(p) = h_o(p, p[0], p[1], \ldots, p[l-1])$$

Note, to minimize hash collisions any *good* hash function should use values of all the fields in an object. Also, to avoid collisions between objects whose values are permutations of each other, the hash function should use the location of the object in the state. For a state $S = p_0, p_1, \ldots, p_n$, there is a hash function h_S that generates the hash for the state using the partial hash values of the objects.

$$\mathcal{H}(S) = h_S(\mathcal{H}(p_0), \mathcal{H}(p_1), \ldots, \mathcal{H}(p_n))$$

Finally, we assume that $\mathcal{H}(S)$ can be incrementally updated. Let $S(p, p')$ represent the state obtained from S by modifying the object $p \in S$ by p' and leaving all other objects unmodified. One of p or p' can be *null* to represent object allocations and deletions respectively. Given $S' = S(p, p')$ assume there is a hash update function h_u to obtain the hash value of S'.

$$\mathcal{H}(S(p, p')) = h_u(\mathcal{H}(S), \mathcal{H}(p), \mathcal{H}(p'))$$

Typically, the update function h_u is computationally much more efficient than h_o or h_S. After a transition, a model checker applies h_o on all object modified in a transition and then uses h_u once for each modified object to determine the hash value of the final state. In effect, the cost of computing the hash value is proportional to the total size of the objects modified in a transition.

In practice, it is very straightforward to design hash functions h_o, h_S, h_u from existing hash functions. See Appendix B for such an example. Also for simplicity, the presentation above assumes that the incremental hashing is performed at the granularity of individual heap objects. However, implementations might choose to perform incremetnal hashing at finer granularities, especially when objects are large. The results in this paper can be easily extended in such cases.

The main goal of this paper is to allow this incremental hash computation when model checkers perform heap canonicalization.

3 Heap Canonicalization

This section describes the need for heap canonicalization when model checking software programs and presents an informal description of the Iosif's algorithm [7].

When a program dynamically allocates objects in the heap, the exact memory location of these objects is arbitrary from the perspective of the program. The memory location of an object is determined by some internal heap allocation algorithm and typically depends on the order of all previous object allocations and deletions.

When a model checker explores different event interleavings during state exploration, it can generate multiple representations of the heap that differ in the memory locations of the objects, but are otherwise equivalent. For example, Figure 1 shows two representations of a linked list with three elements. The two representations differ in the memory locations of the second and the third element in the list. As heap objects can be accessed only through pointers from global variables or other heap objects, no program can differentiate between the two different representations in Figure 1. [3] From the perspective of the model checker, these two representations describe the same state, and thus should explore at most one of them.

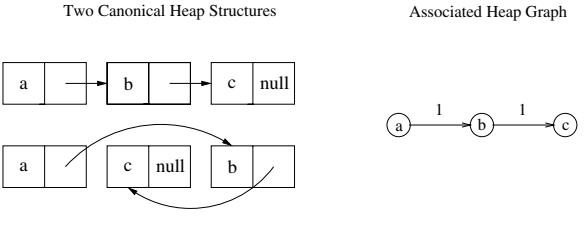


Fig. 1. Two equivalent representations of a linked list, and their common heap graph. This first element of the linked list is the global node.

3.1 The Heap Graph

To formalize the notion of the equivalence of heap states, one can introduce the notion of a heap graph. Informally, a heap graph *abstracts* the memory locations of the heap objects, while maintaining information about the pointers to these objects.

Given a state S, the heap graph $G(S)$ is defined as follows. For each object $p \in S$, $G(S)$ contains a vertex given by $V(p)$. There is a directed edge in $G(S)$ from $V(p)$ to $V(q)$ if and only if p points to q in S. This edge is labeled by the offset of the field in p that contains the pointer to q. For instance, the heap graph

[3] Even in C programs, which are allowed to inspect the value contained in pointers, any behavior that relies on absolute values of the pointers can be safely marked as an error.

of the linked list is shown in Figure 1. Obviously, no two outgoing edges of an object have the same label. Each node in the heap graph is labeled by the values of all the non-pointer fields in the corresponding object as shown in Figure 1.

Propostion 1. *A program cannot differentiate between two states S_1 and S_2 if their heap graphs $G(S_1)$ and $G(S_2)$ are isomorphic.*

Basically, a heap graph captures the entire state of the heap along with all the global variables, but abstracts the memory addresses contained in the pointer variables.

3.2 The Canonical Representation of a Heap

The aim of a heap canonicalization algorithm is to produce the same canonical representation for all heaps that have the same heap graph. By computing the hash on this canonical representation, the model checker can ensure that it explores at most one among the possible equivalent states.

The generation of a canonical representation can be considered as a *relocation* of the objects in the heap. Conceptually, the canonicalization algorithm relocates each object to a *canonical* location determined by the algorithm. After relocation, the algorithm modifies all pointers to an object to reflect this new location.

Formally, a canonicalization algorithm defines a relocation function $reloc : H \rightarrow H$ such that $reloc(p)$ determines the canonical location of an object $p \in S$. We restrict the *reloc* function to have the following desirable properties. First, it is not necessary to relocate the *root* object. Thus,

$$reloc(root) = r, \text{ for a constant } r \tag{1}$$

Second, it is desirable to relocate heap objects in their entirety. In other words, the offset of fields in an object should be invariant during relocation.

$$reloc(p + f) = reloc(p) + f, \quad \forall f : 0 \leq f < len(p) \tag{2}$$

Equations 1 and 2 imply that a particular heap canonicalization algorithm only defines the relocation function for start addresses of heap objects. The two equations determine the relocation for other addresses $\in H$.

Finally, there is a constraint on the relocation function defined by any heap canonicalization algorithm. The *reloc* function should not overlap objects in the canonical heap. That is, for all $p, q \in H$

$$reloc(p) = reloc(q) \iff p = q \tag{3}$$

Given a relocation function that satisfies the above constraints, the canonical representation of a state can be obtained as follows. First, relocate each object $p \in S$ to the location $reloc(p)$. Also, if an object p points to q, modify the pointer value to reflect the new location of q. The values of all non-pointer fields do not change in this relocation. That is,

$$reloc(p)[f] \doteq \begin{cases} reloc(p[f]) & \text{if } p[f] \in H \\ p[f] & \text{otherwise} \end{cases}$$

Abusing notation we will define $reloc(S)$ as the state thus obtained from S.

The goal of a heap canonicalization algorithm is define a relocation function such that $reloc(S) = reloc(S')$ whenever the heap graphs $G(S)$ and $G(S')$ are isomorphic. Given a state S, a model checker computes $reloc(S')$ and inserts the latter in the hash table. Note, it is not necessary to *physically* relocate the objects in the heap. The model checker merely applies the $reloc$ function on all pointer values in the heap when computing the hash.

3.3 Iosif's Algorithm

Iosif's algorithm [7] involves a depth first traversal of the heap graph starting from the *root* object. The traversal uses the edge labels in the heap graph to deterministically order all outgoing edges of a node. [4] The algorithm relocates the heap objects in the heap in the order visited by the traversal. Specifically, if p_i represents the object with depth first order number i, then

$$reloc(p_i) = \sum_{j=0}^{i-1} len(p_j) \tag{4}$$

Note that the above relocation function satisfies the constraint in Equation 3.

Figure 2(a) shows an example. The heap consists of three objects in a binary tree. The *head* node is the *root* object and contains pointers to a *left* node and a *right* node. Assuming that the size of these nodes is 3 word lengths, the algorithm relocates the head, left and right nodes at offsets 0, 3 and 6 respectively in the canonical heap. Also, after relocating the objects, the algorithm modifies the left and right pointers in the head node to point to the respective objects in their new locations.

The correctness of the Iosif's algorithm is as follows. If two heap states have the same heap graph, then the algorithm will visit the heap objects in the same order and relocate them at the same locations. This produces the same canonical representation.

3.4 Need for an Incremental Algorithm

Now we can see why Iosif's algorithm is not suitable for incremental hash computation — it unnecessarily modifies the relocation of objects in the canonical heap. To illustrate this, consider the heap in Figure 2(a) and a transition that deletes the left node in the binary tree. Figure 2(b) shows the resulting heap and the canonical representation generated by the Iosif's algorithm. The algorithm locates the right node at offset 3 in the canonical heap, while the node had an offset 6 before the transition. This change in the location invalidates the precomputed hash value for the right node, even though the node was not modified in the current transition.

[4] Without such an ordering, the heap canonicalization problem is an instance of the graph isomorphism problem, and so is intractable [11].

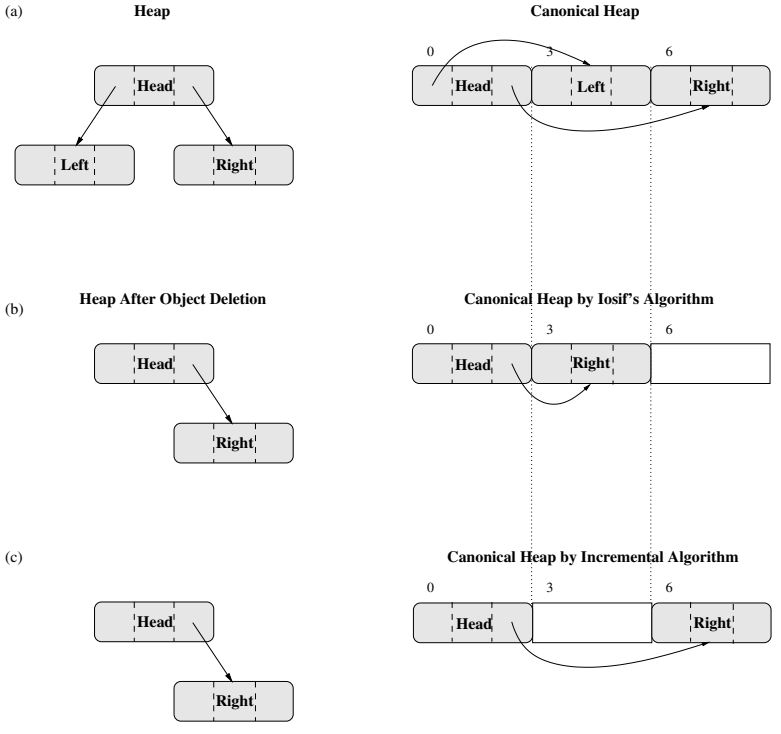


Fig. 2. Demonstration of Heap Canonicalization before and after a transition that deletes a node. The head node is global.

In general, Iosif's algorithm determines $reloc(p)$ from the depth first order number of p in the heap graph (see Equation 4). Any change in the heap graph can potentially modify the relocation of all objects after the change in the depth first order. For instance, one can expect an object addition or deletion to modify on average the relocation of half of all the heap objects. Moreover, whenever $reloc(p)$ changes, the algorithm needs to modify all the pointers to p. This invalidates the precomputed hash for any object that contains such a pointer. In practice, the Iosif's algorithm requires recomputing the hash value for large portions of the heap, drastically slowing the model checker performance.

4 Incremental Heap Canonicalization

This section describes an incremental heap canonicalization algorithm that improves upon the Iosif's algorithm.

The incremental algorithm has two requirements. First and foremost, the algorithm should guarantee heap canonicalization, and thus generate the same canonical heap for all equivalent heaps. Second, the algorithm should seek to reduce unnecessary changes to the canonical representation for small changes in

the heap. Ideally, after every transition the model checker should only need to recompute the hash for objects that are modified in the transition.

To illustrate this, consider the example in Figure 2. When the transition deletes the left node, the incremental algorithm generates a canonical heap as shown in Figure 2(c). Note, the right node remains at offset 6 in the canonical heap both before and after the transition. This enables the model checker to reuse the hash computed for the object before the transition. To guarantee canonicalization, the incremental algorithm should produce the canonical heap in Figure 2(c) for any heap equivalent to the state in Figure 2(c). It is important to note once again that the canonical heap is a conceptual representation used during hash computation – the heap itself is not physically modified. In particular, the empty space between the two objects in Figure 2(c) does *not* represent unreclaimed space in the heap.

The basic idea behind the incremental algorithm is to determine the relocation of an object from the *bfs access chain* of the object, which is a shortest path between the object and the global object. For example, the right node in Figure 2 always has the same bfs access chain that consists of the right pointer from the head node. Thus, the incremental algorithm *always* relocates the right node at the same offset in the canonical heap, irrespective of other objects in the heap. On small changes to a graph, the shortest paths between most objects are likely to remain the same [8,9]. This is specifically so for the heap graph of programs, which typically use large number of data structures that are only weakly related.

4.1 Access Chains

Heap objects can only be accessed through pointers from global variables. For instance, a C program can access the right node in Figure 2(a) with an expression `head->right`. In general, a heap object p can be accessed through a chain of pointers. This access can be represented by the sequence $root = p_0, f_0, p_1, f_1, \ldots, p_n = p$, where f_i is the offset of a field in p_i that contains a pointer to p_{i+1}, for $0 \leq i < n$. In the heap graph, this chain forms a path starting from *root*. This path is uniquely defined by the offset labels on the path edges.

Formally, an *access chain* of a heap object p is a path in the heap graph from the *root* object to p and is denoted by $\langle f_0, f_1, \ldots, f_{n-1} \rangle$, the list of offset labels on the path edges. For instance, the access chain of the third element in the linked list of Figure 1 is $\langle 1, 1 \rangle$, assuming that the first element is the *root* object. Also, the *root* object has an empty access chain $\langle\rangle$.

4.2 BFS Access Chain

A breadth first traversal of the heap graph naturally defines an access chain for objects in the graph. During a breadth first traversal, the edges used to traverse the graph form a spanning tree of the graph rooted at the global node. For any object in the graph, this spanning tree provides a shortest path from the global node to that object. Obviously, the access chain that corresponds to this path is

one of the shortest of all access chains for the object. Additionally, if the breadth first traversal traverses the edges from an object in the increasing order of their offset labels, then the access chain constructed above is guaranteed to be the lexicographically smallest of all shortest access chains of the object.

Formally, a *bfs access chain* of an object p given by $\langle p \rangle$, is the access chain $\langle f_0, f_1, \ldots, f_n \rangle$ such that for any other access chain $\langle g_0, g_1, \ldots, g_m \rangle$ of the object the following holds: either $m > n$; or $m = n$ and there is an $i < n$ such that for all $0 \leq j < i$, $f_j = g_j$ and $f_i < g_i$. The bfs access chain of all objects in the heap graph can be constructed by performing a breadth first traversal of the graph that traverses all outgoing edges of a node in the edge label order. Given a heap graph, the *bfs access chain* of an object is unique. The incremental heap canonicalization algorithm uses this chain to determine the location of objects in the canonical heap.

4.3 Defining the *reloc* Function

A heap canonicalization algorithm is defined by the *reloc* function that determines the location of a heap object p in the canonical representation. The aim of the incremental algorithm is to define *reloc* such that $reloc(p)$ depends *only* on $\langle p \rangle$.

Also, in a type unsafe language such as C, a single pointer field can point to objects of multiple types, and hence multiple lengths. Accordingly, objects of different lengths can have the same bfs access chain. To differentiate such objects in the canonical representation, we also require that $reloc(p)$ depends on $len(p)$. This, as will be explained below, is also crucial to satisfy Equation 3.

In summary, the incremental algorithm seeks for a function *canon* that takes $\langle p \rangle$ in a suitable encoding and $len(p)$ as arguments, and returns $reloc(p)$.

$$reloc(p) \equiv canon(\langle p \rangle,\ len(p)) \tag{5}$$

The next task is to obtain an encoding for $\langle p \rangle$. Let $\langle p \rangle = \langle f_0, f_1, \ldots, f_n \rangle$. If $parent(p)$ is the parent of the object p in the breadth first traversal of the heap graph, then $\langle parent(p) \rangle = \langle f_0, f_1, \ldots, f_{n-1} \rangle$. This hints at an encoding for $\langle p \rangle$ from the encoding for $\langle parent(p) \rangle$ and f_n. Using Equation 5 recursively, the following theorem shows that $reloc(parent(p) + f_n)$ provides the necessary encoding for $\langle p \rangle$.

Theorem 1. *For any arbitrary function canon* : $H \times N \rightarrow H$ *and for a heap object p, the relocation function defined by*

$$reloc(p) \doteq canon(reloc(parent(p) + f_n),\ len(p)) \tag{6}$$

relocates two objects with the same bfs access chain and the same length to the same location in the canonical heap.

Proof: The proof follows from a simple induction on the depth of the bfs access chain of an object. Note, the theorem only specifies the relocation function for

the start address of heap objects. The relocation of other addresses in H is automatically given by Equations 1 and 2.

The global object has an empty bfs access chain and the base step trivially follows from Equation 1. Assume the theorem holds for all objects with a bfs access chain of length $\leq n$. Consider two objects p_1, p_2 such that they have the same length and the same bfs access chain. Let

$$\langle p_1 \rangle = \langle p_2 \rangle = \langle f_0, f_1, \ldots, f_n \rangle$$

Obviously, $parent(p_1)$ and $parent(p_2)$ have the same bfs access chain of length n.

$$\langle parent(p_1) \rangle = \langle parent(p_2) \rangle = \langle f_0, f_1, \ldots, f_{n-1} \rangle$$

By induction

$$reloc(parent(p_1)) = reloc(parent(p_2))$$

Using Equation 2,

$$reloc(parent(p_1) + f_n) = reloc(parent(p_2) + f_n)$$

This from Equation 6 and the fact that $len(p_1) = len(p_2)$ implies that

$$reloc(p_1) = reloc(p_2)$$

□

Note that f_n is the offset of the field in $parent(p)$ that points to p, and thus $reloc(parent(p) + f_n)$ represents the address of that field in the canonical heap. Theorem 1 essentially shows that $\langle p \rangle$ can be captured by this address in the canonical heap. This greatly simplifies the implementation of the incremental canonicalization algorithm.

The following theorem follows.

Theorem 2. *For an arbitrary function canon* : $H \times N \rightarrow H$*, the relocation function defined by Equation 6 generates the same canonical representation for two heaps with the same heap graph, provided Equation 3 holds.*

Proof: The bfs access chain is an inherent property of an object in the heap graph. Thus, given two heaps with the same heap graph, equivalent objects in the two heaps have the same bfs access chain. The proof follows from Theorem 1. □

4.4 Designing the *canon* Function

By Theorem 2, heap canonicalization is guaranteed for *any* function *canon* : $H \times N \rightarrow H$ provided the resulting *reloc* function satisfies the constraint in Equation 3. The final task is to design one such function.

Without apriori knowing the length of the objects with different bfs access chains, a closed form for the *canon* function is not possible. The trick then, is to define the *canon* function *incrementally* during model checking. The algorithm

is shown in Figure 3 and implements *canon* as a hash table. Initially, the hash table is empty. When the value for $canon(addr, len)$ is required for a new address length pair, the algorithm *allocates* a new region in the canonical heap of the required length. This region is never reused for other address length pairs. Thus no overlap is possible in the canonical heap, satisfying Equation 3.

The *canon* hash table is a global table maintained by the model checker. During state exploration, the model checker can add new entries to the table, but can never modify an existing entry or delete it. This ensures that the hash table implements a *function* which is essential from Theorem 2 for heap canonicalization. Also, the size of the *canon* table grows as the model checker discovers different bfs access chains in the program. However, the number of such chains tends to stabilize once the data structures are initialized in the program. In our experiments (§5), the *canon* table did not exceed ten thousand entries.

As a demonstration of the algorithm, consider the example in Figure 2. After processing the state in Figure 2(a), the *canon* table maintains the following mapping: $(0, 3) \rightarrow 3$ and $(2, 3) \rightarrow 6$. In other words, the *canon* table remembers that an object of length 3 pointed from the offset 2 (the right pointer) of the head node should be relocated at offset 6 in the canonical heap. Now, when the left node is deleted, the mapping maintained above produces a canonical representation in Figure 2(c) as desired.

5 Experimental Results

We have evaluated the incremental heap canonicalization algorithm in two explicit model checkers, CMC [12] and Zing [10]. CMC is specifically designed for checking network protocol implementations. It executes the implementation directly without resorting to any intermediate representation. A transition involves the entire processing of a protocol event, such as packet receives or timer interrupts, and can typically involve more than tens of thousands of instructions. Zing focuses on detecting concurrency errors in large software programs. The input language is designed for automatic translation of software programs into Zing models, and provides support for dynamic object and thread creation. Zing explores thread interleavings at much finer granularity than CMC, and a transition in Zing typically involves few instructions.

CMC and Zing represent two fundamentally different model checkers, and the incremental heap canonicalization algorithm performs well in both of them. Table 1 shows the improvement achieved by the model checkers on three large models. As the state sizes grow, the incremental algorithm fares much better than the non-incremental version. In these examples, the model checker processes only 5% of the heap per transition. As a result, the model checker runs 2 to 9 times faster.

Table 1. Performance improvement of the Incremental Canonicalization Algorithm

Model	State Size (KB)	No. of Heap Objects (avg)	% Modified per Transition	% Accessed for Hashing	Model Checker Speedup
Tx. Manager (Zing)	1.3	46.0	0.16	4.25	x2.15
File System (Zing)	4.1	364.1	0.4	2.18	x3.59
Linux TCP (CMC)	255.7	102.7	4.96	5.06	x8.85

In the Linux TCP case, the incremental algorithm almost always processes only the objects that are modified by the system. However, this is not the case with the Zing models. We believe that this is due to a deficiency of the algorithm implementation in Zing. Specifically, the algorithm does not process the thread stack frames incrementally. We hope to rectify this very soon.

While running our experiments, we found that both CMC and Zing spend their time in performing the breadth-first traversal of the heap required for the incremental algorithm, and *not* on the hash computation. However, note that such a full-heap traversal is required even to check for memory leaks (or to perform garbage collection). One way to avoid this full-heap traversal is to implement an incremental shortest path algorithm, such as [13], for computing the bfs access chains and for detecting memory leaks. We hope to pursue this approach in our future work.

6 Related Work

The work presented in this paper is related to, and in many ways relies on the observations made in [6,7]. The need for heap canonicalization was first observed by Lerda et.al. [6]. However, they only provide a heuristic algorithm that does not guarantee uniqueness of the canonical representation for all equivalent heaps. Iosif's algorithm [7] is the only previously known heap canonicalization algorithm to guarantee this uniqueness. This algorithm has been extended to support thread symmetries [14]. Extending the incremental heap canonicalization algorithm to recognize thread symmetries is interesting future work.

Other researchers have observed the expensiveness of state hashing in explicit model checking [5]. In [15], Dillinger et.al. provide a method to reduce the *number* of hash functions required for bitstate hashing. This approach is orthogonal to the approach presented in the paper and can be used together.

7 Conclusions

This paper presents an incremental heap canonicalization algorithm that is necessary when explicitly model checking large software programs. As demonstrated by the experiments, the incremental algorithm scales well to large heaps by generating the canonical representation of a heap incrementally.

References

1. Holzmann, G.J.: From code to models, Newcastle upon Tyne, U.K. (2001) 3–10
2. Corbett, J., Dwyer, M., Hatcliff, J., Laubach, S., Pasareanu, C., Robby, Zheng, H.: Bandera: Extracting finite-state models from java source code. In: ICSE 2000. (2000)
3. Brat, G., Havelund, K., Park, S., Visser, W.: Model checking programs. In: IEEE International Conference on Automated Software Engineering (ASE). (2000)
4. Musuvathi, M., Park, D., Chou, A., Engler, D.R., Dill, D.L.: CMC: A Pragmatic Approach to Model Checking Real Code. In: Proceedings of the Fifth Symposium on Operating Systems Design and Implementation. (2002)
5. Holzmann, G.J.: The Spin Model Checker: Primer and Reference Manual. Addison Wesley, Boston, Massachusetts (2003)
6. Lerda, F., Visser, W.: Addressing dynamic issues of program model checking. Lecture Notes in Computer Science **2057** (2001) 80–102
7. Iosif, R.: Exploiting Heap Symmetries in Explicit-State Model Checking of Software. In: Proceedings of 16th IEEE Conference on Automated Software Engineering. (2001)
8. Frigioni, D., Marchetti-Spaccamela, A., Nanni, U.: Incremental algorithms for single-source shortest path trees. In: Proceedings of Foundations of Software Technology and Theoretical Computer Science. (1994) 112–224
9. Narvaez, P., Siu, K.Y., Tzeng, H.Y.: New dynamic SPT algorithm based on a ball-and-string model. In: INFOCOM (2). (1999) 973–981
10. Andrews, T., Qadeer, S., Rehof, J., Rajamani, S.K., Xie, Y.: Zing: Exploiting program structure for model checking concurrent software. In: Proceedings of the 15th International Conference on Concurrency Theory. (2004)
11. Gary, M.R., Johnson, D.S. In: Computers and Intractability. Freeman (1979)
12. Musuvathi, M., Park, D., Chou, A., Engler, D., Dill, D.: CMC: A pragmatic approach to model checking real code. In: Proceedings of Operating Systems Design and Implementation (OSDI). (2002)
13. Ramalingam, G., Reps, T.W.: An incremental algorithm for a generalization of the shortest-path problem. J. Algorithms **21** (1996) 267–305
14. Robby, Dwyer, M.B., Hatcliff, J., Iosif, R.: Space-reduction strategies for model checking dynamic software. In: SoftMC03 Workshop on Software Model Checking, Electronic Notes in Theoretical Computer Science. Volume 89. (2003)
15. Dillinger, P.C., Manolios, P.: Bloom filters in probabilistic verification. In: Formal Methods in Computer-Aided Design (FMCAD). (2004)
16. Carter, J.L., Wegman, M.N.: Universal classes of hash functions. In: Journal of Computing and System Sciences. (1979) 143–154
17. Cormen, T.H., Leiserson, C.L., Rivest, R.L. In: Introduction to Algorithms. MIT Press (1990)

A Implementation of the *canon* Function

See Figure 3

```
canon_table; //implemented as a hash table
unalloc_address = canonical_heap_address_start;

canon(parent_ptr, len){
  if (canon_table[parent_ptr, len] is defined){
     return canon_table[parent_ptr, len];
  }
  else{
    // incrementally define canon for (parent_ptr, len)
    canon_table[parent_ptr, len] = unalloc_address;
    unalloc_address += len;
    return canon_table[parent_ptr, len];
  }
}
```

Fig. 3. The implementation of the canon function

B Example of an Incremental Hash Function

One example that is shown below is the universal hash function [16] described in [17]. Given a state x consisting of n bytes $x = \{x_1, \ldots, x_n\}$, a hash table of size m where m is prime, and a random array r of n elements $r = \{r_1, \ldots, r_n\}$ such that $0 \leq r_i \leq m-1$, the hash function is given by

$$h(x) = \sum_{i=1}^{n} r_i x_i \pmod{m}$$

An object p of length l in the state consists of the following bytes $\{x_p, x_{p+1}, \ldots, x_{p+l-1}\}$. Thus, its partial hash value is the following partial sum.

$$\mathcal{H}(p) = \sum_{i=p}^{p+l-1} r_i x_i \pmod{m}$$

The hash value of the entire state is the sum (modulo m) of the partial hash valuse of all objects in the state. The hash update function is simply

$$\mathcal{H}(S(p, p')) = \mathcal{H}(S) - \mathcal{H}(p) + \mathcal{H}(p') \pmod{m}$$

Memory Efficient State Space Storage in Explicit Software Model Checking

Sami Evangelista and Jean-François Pradat-Peyre

CEDRIC - CNAM Paris,
292, rue St Martin, 75003 Paris
{evangeli, peyre}@cnam.fr

Abstract. The limited amount of memory is the major bottleneck in model checking tools based on an explicit states enumeration. In this context, techniques allowing an efficient representation of the states are precious. We present in this paper a novel approach which enables to store the state space in a compact way. Though it belongs to the family of explicit storage methods, we qualify it as semi-explicit since all states are not explicitly represented in the state space. Our experiments report a memory reduction ratio up to 95% with only a tripling of the computing time in the worst case.

1 Introduction

Model checking is a powerful and automatic technique for the verification of finite state systems. It consists of enumerating all the possible configurations (states) or actions of the system to track the ones which do not match the specification, usually expressed in a temporal logic, e.g. LTL. To preserve termination and to improve efficiency, model checking algorithms have to keep track of the visited states in a state space (or reachability set).

Algorithms based on an explicit state enumeration suffer from the well-known state explosion problem. Due to the concurrent execution of several components, the number of possible configurations can be far too large to fit in memory or even on a disk. In a first family of techniques designed to alleviate this problem, we can put all those which aim at reducing the number of visited states while still preserving the property to verify. Examples of such techniques include partial order reductions and symmetry-based reductions. A second family of techniques aim at representing the state space in an efficient way in order to limit the memory allocated to store the state space. State compression techniques and state caching are examples of such techniques.

Methods which belong to the first family usually achieve better reductions of the amount of memory required since, in most systems, the number of visited states is an exponential function of the number of concurrent process. However, a wise choice for the representation of the state space can still perform a significant reduction of the memory requirement. Moreover, the use of such techniques is especially needed when model checking is applied to software specification or

P. Godefroid (Ed.): SPIN 2005, LNCS 3639, pp. 43–57, 2005.

models extracted from source code, since the state descriptor of these models can grow very large as it may represent complex data such as heap allocated objects.

This paper presents a compression method which falls into the second family. We qualify our method as semi-explicit since states may not be explicitly represented in the state space. The underlying idea of the method is conceptually simple but reveals to be very useful in practice. Moreover, it is not specific to the formalism used in this paper (colored Petri nets) and it could easily be adapted to other formalisms. Thus we formulate our ideas in general terms. Last but not least, our technique can be further combined with the state collapsing method [1, 2] to lead to better state representations.

The remainder of this work is organized as follows. Section 2 is a brief overview of the methods proposed so far to represent the state space in explicit state model checking. Section 3 recalls some basic concepts of colored Petri nets. Our storage method is presented in Section 4. The results of some experiments made with our model checker are presented in Section 5. Section 6 discusses the compatibility of our method with other techniques used in explicit state model checking and presents some directions for future research. Lastly, Section 7 presents our conclusions.

2 State Space Representation in Explicit Model Checking

This section draws up an overview of the methods used to represent the state space in explicit model checking. These methods can be classified in three categories: exhaustive storage, partial storage, and lossy storage. The method we propose in this paper belongs to the first family.

Exhaustive storage methods build the complete state space of the system. Each state met during the search is encoded by a reversible mapping into a state vector and then stored in an adequate data structure, e.g., a hash table. In this way, each state can be retrieved from its encoded form and checking whether a state has already been visited becomes trivial. Compression techniques are usually employed to optimize the encoding. Examples of compression techniques are state collapsing [1], recursive indexing [2] (possibly improved by training runs [2]), very tight hashing [3], sharing trees [4], difference compression [5], runtime state compaction [6], or invariant-based compression methods for Petri nets as it is done in [7] and [8]. Each method tries to provide efficient compression ratios without introducing an unbearable increase in the running time.

Partial storage methods are based on the idea that it is not necessary to store all the visited states to preserve the termination of the search algorithm, but rather a portion of the state space. Perhaps the most famous method that follows this paradigm is the state space caching method [9]. In state space caching, only the states which belong to the search stack are stored in the state space in order to

avoid entering cycles which would endanger the termination of the algorithm. Some other states may also be stored depending on the amount of memory still available. The main limitation of state caching is the amount of redundant work performed. Indeed, if only stack states are stored, each state s will be explored once for every path leading from the initial state to s, causing an unacceptable blowup in the execution time. It was observed in [9] that the use of partial order methods greatly reduces this problem by limiting the exploration of redundant paths. The main drawback of caching techniques is that the best replacement strategy to adopt and the run time increase heavily depends on the structure of the state space, and is thus hard to predict as observed in [10].

In [11], the authors proposed to improve reachability analysis via the use of pseudo-root states. These states are characterized by the fact that all their predecessors have already been visited during the search. This gives the possibility to forget these states without endangering the termination of the search. Experiments reported result in 2- to 16-fold improvement in space requirements. This technique has the advantage of introducing little runtime overhead, since any state is only visited once. Nevertheless, it relies on the ability to compute predecessors of states which seems to us a strong requirement.

The sweep line method, another technique based on this observation, was recently introduced in [12, 13]. It is inspired from the garbage collection mechanism. It uses the notion of progress present in some systems, e.g., timed protocols, to safely delete from the state space the states which cannot be reached again from the set of unprocessed states. Preliminary results on this method are quite promising. However, it suffers from the fact that a progress measure function has to be supplied by the user. In [14] Schmidt gave a method to automatically derive such a function from the incidence matrix of Petri nets, but it seems hard to construct a "good" function for more complex formalisms.

More recently, Behrmann, Larsen and Pelánek proposed in [15] several strategies to decide whether or not a state has to be kept in the reachability set while still preserving termination and efficiency.

Verisoft [16] is a software model checker that illustrates the partial storage strategy to the extreme. Verisoft does not store any states at all. The termination is guaranteed by limiting the search to a specified depth. It makes use of partial order methods (sleep sets) to avoid multiple revisits of the same state.

Lossy storage In a lossy storage scheme, each state is mapped to a state vector in a non-injective way before its insertion into the state space. Two different states may thus share the same compressed representation. An example of technique that belongs to this family is the bitstate hashing technique of Holzmann (or supertrace) [17]. Lossy storage methods greatly cut down the memory requirement since an arbitrary small number of bits can be used to represent each state, but suffer from an omission probability that a state may be erroneously declared as already visited whereas it is new, leading to unexplored portions of the state space. Wolper and Leroy showed in [18] that this probability can be reduced by storing the whole hash signature in a hash table with conflict resolution (the hashcompact method), or by using multiple hash functions (the

multihash method). However, this kind of technique cannot be used for verification purposes, but rather at a debugging stage since the omission probability cannot be reduced to 0.

3 Colored Petri Nets

We develop our method in the context of colored Petri nets [19]. Colored Petri nets allow the modeling of complex systems in a compact way and support numerous analysis techniques. In a colored Petri net, a place contains typed (or colored) tokens instead of anonymous tokens of Petri nets, and a transition may be fired in multiple ways, i.e., instantiated. To each place and each transition is attached a type (or a color domain). Each arc of the net between a place and a transition is labeled by a color mapping which specifies the type and the number of tokens produced or consumed by the firing of the transition for a given instantiation.

The definition of colored Petri nets is based on multi-sets. A multi-set over a set S is a mapping from S to the set of positive integers. The set of multi-sets over a set S is noted $Bag(S)$. Addition, subtraction, and comparison of multi-sets are defined as usual. States of colored Petri nets are also called markings.

Definition 1. *A colored Petri net is a tuple* $N = \langle P, T, \Sigma, C, W^-, W^+, m_0 \rangle$ *where* P *is a finite set of* **places***;* T *is a finite set of* **transitions***, with* $P \cap T = \emptyset$*;* Σ*, the* **colors set***, is a finite set of finite and non empty sets;* C*, the* **color domain** *application, is a mapping from* $P \cup T$ *to* Σ*;* W^- *and* W^+*, the* **backward and forward incidence matrixes** *associate to each* $(p, t) \in P \times T$ *a color mapping from* $C(t)$ *to* $Bag(C(p))$*; and* m_0*, an* **initial marking** *is an element of* $\mathbb{M}_N$*, the set of mappings which associate each* $p \in P$ *to an element of* $Bag(C(p))$*.*

We now define the transition relation and the state space of colored Petri nets.

Definition 2. *Let* $N = \langle P, T, \Sigma, C, W^-, W^+, m_0 \rangle$ *be a colored Petri net,* $t \in T, c_t \in C(t)$ *and* $m \in \mathbb{M}_N$*. The transition instance* c_t *of* t*, noted* (t, c_t) *is* **firable***, at* m *(noted* $m[(t, c_t)\rangle$*) if and only if* $\forall p \in P, m(p) \geq W^-(p, t)(c_t)$*. The* **firing** *of* (t, c_t) *at* m *leads to a marking* m'*, (noted* $m[(t, c_t)\rangle m'$*) defined by* $\forall p \in P, m'(p) = m(p) - W^-(p, t)(c_t) + W^+(p, t)(c_t)$*. The* **state space** *(or reachability set) of* N*, denoted by* $\mathbb{R}_N$*, is defined recursively as the set* $\{m_0\} \cup \{m \in \mathbb{M}_N | \exists m' \in \mathbb{R}_N, t \in T, c_t \in C(t) | m'[(t, c_t)\rangle m\}$*.*

4 The Δ-Markings Storage Method

This section describes the Δ-markings method and is organized as follows. Firstly, we present the general idea of the method. In a second step we give a depth first search (DFS for short) algorithm based on it. Two direct optimizations of the algorithm are then described. It is then shown how our method can be combined with the state collapsing compression method to obtain very compact representations. Lastly, we close the section with a short analysis of the method.

General idea. In colored Petri nets, as in many other formalisms, the transition relation is a deterministic mechanism: the firing of a transition instance at a marking leads to a single marking. On the basis of this determinism, we propose to store some markings of the reachability set in a non explicit way: instead of storing the actual value of a marking m, we only store a reference to one of its predecessors m' and a transition instance (t, c_t) whose execution leads from m' to m. Because of the determinism of the transition relation, this representation of m' is unambiguous although it is not canonical since a marking may have several predecessors. Markings stored in this manner are called Δ-markings and are said to be stored symbolically while markings stored in the usual way are said to be stored explicitly.

Storing a reference to a marking and a transition instance should obviously lead to better state representations, especially when the modeled system exhibits large state vectors. However, this representation presents a drawback: the test for checking whether or not a marking m is new or not can be significantly slowed. This test usually entails comparing m to some marking(s) m' stored in the state space. In the classical scheme, m' is stored as a vector of bits and so is encoded m before its insertion into the state space. The comparison can then be efficiently implemented by a bits vectors comparison. When the reachability set contains Δ-markings, the operation is more complicated. Let us assume that we have a sequence of markings $m_1, m_2, \ldots, m_n = m'$ such that m_1 is stored explicitly and each $m_i \neq m_1$ is stored as a Δ-marking which points to m_{i-1} with the binding (t_{i-1}, c_{i-1}) such that $m_{i-1}[(t_{i-1}, c_{i-1})\rangle m_i$. The idea is then to backtrack to m_1, and to apply to it the firing of bindings sequence $(t_1, c_1).(t_2, c_2)\ldots(t_{n-1}, c_{n-1})$ to have an "explicit view" of m'. Once this operation realized, the comparison of m and m' becomes straightforward.

We will call a *reconstitution* the operation which consists in finding the actual value from a Δ encoding, and the sequence of transition bindings which enables to reconstitute a marking will be called a *reconstituting sequence*. The principle of the reconstitution mechanism can be illustrated with the help of Figure 1. Let us suppose, for instance, that we have to reconstitute marking m. To do so, we will first have to backtrack to m'. Since it is not stored explicitly, we will then have to backtrack to m_0 and finally apply to it the reconstituting sequence $(t', c').(t, c)$. This operation allows us to retrieve the actual value of marking m.

The run time overhead caused by the method directly depends on the lengths of the reconstituting sequences that are fired when the algorithm has to determine whether a marking is new or not. In order to place an upper bound on the length of these sequences we use the underlying idea of the stratified caching strategy [10]. We use a parameter k_δ defined by the user which belongs to set of positive integers. During the exploration of the state space, each marking met at a depth d such that $d \mod k_\delta = 0$ is stored explicitly. All other markings are stored symbolically and point to one of their predecessors (by a doted arc on the figure). By this way, we can guarantee that the length of each reconstituting sequence is bounded by $k_\delta - 1$. This idea is illustrated by Figure 1.

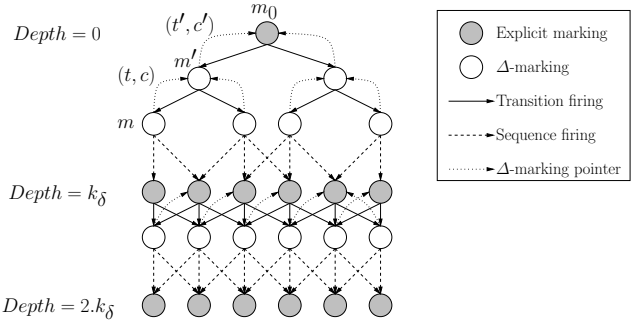


Fig. 1. A state space with Δ-markings

The algorithm. We propose a search algorithm based on our method. Let us point out that we arbitrarily choose to present here a depth first search algorithm but that our method is not specific to this kind of search. The algorithm is given in a pseudo code form in Figure 2. For the sake of simplicity, no distinction is done between a transition and a transition instance.

Markings are stored in a hash table S using a hash function noted HASH. To manage collisions, each slot of the hash table contains a list of markings. When a marking m is encountered we compare it to each marking m' of the list contained in the slot HASH(m) to check whether it is new or not. To achieve this, the function RECONSTITUTE is used to reconstitute a marking stored in the hash table. It recursively backtracks to a marking stored explicitly and apply to it the correct reconstituting sequence. Finally, if m is not in the hash table, it is added to the list of slot HASH(m). Depending on its depth, it is stored explicitly or symbolically. If it is stored symbolically, it points to its predecessor in the search stack. To achieve this, three additional parameters are passed to the DFS procedure: the depth of the marking to explore, the predecessor of the marking in the search stack, and the transition which leads from the predecessor to the marking.

We can easily prove that the reconstitution function RECONSTITUTE terminates, i.e., it cannot enter in a cycle of Δ-markings, since each Δ-marking points to its predecessor in the search stack and each k_δ^{th} marking in the search stack is stored explicitly.

Speeding up the reconstitution process. An efficient implementation of the reconstitution process is crucial for the performances of our method. Moreover, this makes usable the Δ-markings method with higher values of k_δ, leading to very condensed state spaces. We propose now two optimizations that aim at reducing the reconstitution times. Several experiments, that are not reported in this paper for space constraints, showed that the combination of both optimizations leads to an average reduction of the run time of 60%.

```
PROCEDURE DFS (marking m, int depth, stored_marking pred, transition t)
1   h ← HASH(m)
2   for i ∈ [1..LENGTH(S[h])] do
3     if m = RECONSTITUTE(ITH(S[h], i)) then return end if
4   end for
5   if depth = 0 then
6     new.type ← explicit
7     new.m ← m
8   else
9     new.type ← delta
10    new.pred ← pred
11    new.t ← t
12  end if
13  ADD(S[h], new)
14  for t ∈ ENABLED(m) do
15    let m' be such that m[t⟩m'
16    DFS(m', (depth + 1) mod k_δ, new, t)
17  end for
FUNCTION RECONSTITUTE (stored_marking sm)
1   if sm.type = explicit then r ← sm.m
2   else
3     m ← RECONSTITUTE(sm.pred)
4     let r be such that m[sm.t⟩r
5   end if
6   return r
PROCEDURE EXPLORE_STATE_SPACE ()
1   INIT(S)
2   DFS(m_0, 0, nil, nil)
```

Fig. 2. A DFS algorithm based on the Δ-markings method

Updating the predecessors of Δ-markings. The idea of this first optimization, illustrated by Figure 3, is to update the predecessor of a Δ-marking when a shorter path to an explicit marking is found. Let us suppose that the search algorithm successively visits a sequence of markings $m_1[t_1\rangle m_2[t_2\rangle \dots [t_{n-1}\rangle m_n$ and that m_1 is its only marking stored explicitly by the algorithm. Each $m_i \neq m_1$ points to m_{i-1}. When m_n has to be reconstituted, we have to backtrack to m_1 and apply to it the reconstituting sequence $t_1.t_2 \dots t_{n-1}$. Let us suppose that the algorithm visits later a sequence of markings $m[t\rangle \dots [t'\rangle m'[t''\rangle m_n$ shorter than the initial one and such that m is its only marking stored explicitly. We can update the predecessor of m_n and set it to m'. As a consequence, each time the algorithm will reconstitute m_n it will backtrack to m and apply a shorter reconstituting sequence. Not only the reconstitutions of m_n will be sped up, but also the reconstitutions of all the Δ-markings which point to m_n (the set S on the figure).

Several experiments pointed out that the speed improvement is proportional to the parameter k_δ which is not surprising. Nevertheless, we expect that the

benefits obtained with this optimization decrease if some partial order technique is used in combination with our storage method. Since the goal of partial order methods is to reduce the exploration of redundant paths, the possibilities of updating the predecessor of a Δ-marking should naturally be reduced.

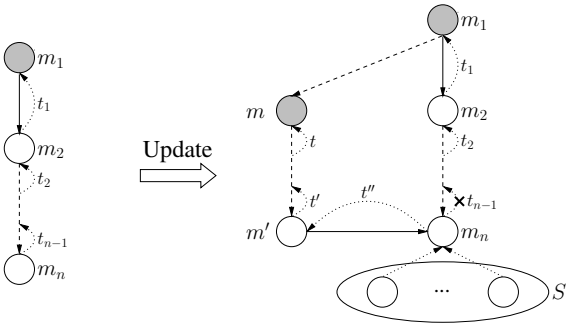


Fig. 3. Updating the predecessor of m_n

Backward firing of the reconstituting sequence. This second optimization is more an implementation trick, but it reveals to be very efficient in terms of reduction of the execution time.

The reconstitution of a Δ-marking δ involves two costly operations: the decoding of the explicit marking e which enables to reconstitute δ, and the firing of the reconstituting sequence σ. In comparison, in a "classical" storage scheme, checking that two markings are equal is simply done by comparing two vectors of bits. However, this reconstitution can be avoided by performing a backward firing, i.e. an "unfiring", of sequence σ. Instead of backtracking to e and firing σ, the idea is to start from m and to unfire σ on it, i.e., find the marking m' which is such that $m'[\sigma\rangle m$. Finally, δ and m correspond to the same marking if and only if $m' = e$. If at some step of the unfiring of σ, let us say after the unfiring of σ_2 such that $\sigma = \sigma_1.\sigma_2$ we obtain a marking m'' which is such that $m''(p) < 0$ for some place p of the net, then this means that the unfiring of σ is not possible at m. We can therefore claim that the reconstitution of δ does not produce the marking m. The full backtrack to e is thus avoided, as well as the decoding of e and the unfiring of σ_1.

The possibility to reverse the transition relation is a prerequisite for this optimization. This one is met for colored Petri nets but it would be more problematic to adapt this optimization to formalisms which do not provide such a facility.

The function RECONSTITUTE of Figure 2 can thus be replaced by function UNFIRE_AND_CHECK given below. The if-statement condition at line 3 of the DFS procedure must naturally replaced by UNFIRE_AND_CHECK(ITH($S[h], i$), m).

```
FUNCTION UNFIRE_AND_CHECK (stored_marking sm, marking m)
1   if sm.type = explicit then r ← sm.m = m
2   else
3     let m' be such that m'[sm.t⟩m
4     if m'(p) < 0 for some place p then r ← false
5     else r ← UNFIRE_AND_CHECK(sm.pred, m')
6     end if
7   end if
8   return r
```

A state collapsing compression scheme for colored Petri nets. The state collapsing method [1] and its improvement, the recursive indexing method [2] are two state compression methods. Both are based on an assumption particularly adapted to software model checking. Thus, this technique is implemented in Bogor [20], JPF [21] and Spin [22]. This hypothesis is the following: even when the number n of syntactically possible states for a process is huge, the number m of states effectively reached by this process is usually much smaller. The idea is then to represent local process states on $log_2(m)$ bits instead of $log_2(n)$ bits. These $log_2(m)$ bits form an index of a table which is shared by all the global states and in which the actual local states are stored on $log_2(n)$ bits. Such a strategy allows to save a significant amount of memory when $n >> m$.

For colored Petri nets, we can adapt this principle on the basis of two observations:

1. Given a place, some token values within its domain may never be held in it.
2. Given a transition, some of its instances may never be firable.

This is so because the types, i.e., color domains, of the places and transitions of the net are usually over-approximations made by the user of the possible values really met during the search.

The first observation can help us to define an efficient compression function for the markings which are stored explicitly. If we note m the number of bits used to store a collapsed item of a color domain, each token value of a place p can be represented with m bits instead of using $log_2(|C(p)|)$ bits.

The second observation is useful to compress Δ-markings. By using the same parameter m, a collapsed Δ-marking will fit in

$$1 + \log_2(H) + \log_2(L) + \log_2(|T|) + m$$

bits. The first bit is used to indicate to the decoder the type of marking to decode. The "pointer" to the predecessor of the Δ-marking in the hash table is a couple (h, i) where h is the slot of the hash table which contains the predecessor, and i is its position in the slot list. If we note H the size of the hash table and L the maximal length of a slot list, $\log_2(H) + \log_2(L)$ bits are sufficient to encode this couple. At last, $\log_2(|T|) + m$ is the number of bits used to encode a collapsed transition instance (t, c_t). Without collapsing the vector, $\log_2(|T|) + \log_2(|C(t)|)$ bits are required to encode this couple.

Since m and L cannot be known before the search terminates, they must be fixed by the user. When the supplied values are not sufficient, an error is reported at the run time to the user who, in turn, can change these parameters and rerun the search. On all the experimentations we made, $L = 2^8$ and $m = 16$ were both sufficient to encode any Δ-marking. With these values each Δ-marking can be represented with approximatively 64 bits, which is the number of bits recommended in [18] to store a hash signature of a state descriptor. Thus, by combining our method with a state collapsing compression scheme we obtain a reliable storage method that performs a reduction similar to the one obtained by an unreliable method such as hashcompact.

Analysis of the method. We address now the following question: "what can we expect from the method in terms of memory reduction ?". If we note N_e the number of markings stored explicitly, N_δ the number of Δ-markings, $N = N_e + N_\delta$, V_e the average size of markings stored explicitly, and V_δ the average size of Δ-markings, the total amount of memory required to store the state space will be $N_e \times V_e + N_\delta \times V_\delta$.

Now if we suppose that the proportion of markings met at depths $0, k_\delta, \ldots, n.k_\delta$ is $\frac{N}{k_\delta}$, we can approximate it by $(\frac{1}{k_\delta} \times N).V_e + (\frac{k_\delta - 1}{k_\delta} \times N).V_\delta$.

By collapsing Δ-markings, we have seen that V_δ becomes a constant value, typically between 8 and 12 bytes. So, for large values of k_δ, e.g., ≥ 50, the memory allocated weakly depends on the size of the state vector and can be approximated by $N \times V_\delta$.

5 Experimental Results

The storage method introduced in this paper, as well as the state collapsing scheme presented in the previous section have both been implemented in Helena [23], an explicit model checker for high level Petri nets. This section presents the results of some experiments that have been made with Helena.

All the experiments described in this section have been made on a Pentium 4, 2.8 Ghz with 2 Gb of RAM. The search method used was a depth first search and the two optimizations previously presented were both turned on.

Experimentation on academic examples. We first consider the six following examples frequently used in benchmarks: the distributed database system, the sieves of Eratosthene, the mutual exclusion algorithm of Peterson, a simple mutual exclusion protocol, the leader election protocol of Chang and Roberts, and lastly the dining philosophers. All these models can be found in the Helena distribution available at `http://helena.cnam.fr`. The results of our experimentations are reported in table 1. For each model, its name and its parameter are given in the first column. Four searches were done with k_δ in the set $\{5, 10, 20, 50\}$. For each of these runs, the relative performance in time and in

space with respect to a classical DFS without state compression are reported respectively in columns T and S. The last row of the table contains the average values observed. The collapse method was not helpful for these simple models and therefore disabled.

Table 1. Results obtained for academic examples

		$k_\delta = 5$		$k_\delta = 10$		$k_\delta = 20$		$k_\delta = 50$	
	States	T	S	T	S	T	S	T	S
Dbm, 12	2 125 765	109%	24%	119%	16%	171%	5%	173%	4%
Eratos, 70	3 177 699	123%	40%	148%	33%	198%	29%	340%	27%
Peterson, 4	3 407 946	131%	64%	151%	60%	191%	58%	316%	57%
Mutex, 17	5 701 632	130%	58%	148%	53%	176%	50%	249%	48%
Leader, 16	10 475 430	137%	46%	176%	39%	231%	37%	329%	32%
Dining, 13	14 741 195	137%	54%	155%	48%	181%	45%	218%	44%
		128%	**56%**	**150%**	**42%**	**191%**	**37%**	**271%**	**35%**

We observe that the best compression ratio provided by our method are obtained for the two models which exhibit the largest state vectors, namely the distributed database system, and the sieves of Eratosthene. The best reduction is obtained for the distributed database system with $k_\delta = 50$. In this case, the average size of the state vector can be reduced from 148 bytes to 8 bytes. Let us note that, for this model, the size of the search stack is bounded by $2 \cdot N$. This explains why the results obtained for $k_\delta = 20$ and $k_\delta = 50$ are so close. In the opposite, for models with small state vectors, the gains obtained are quite smaller. For instance, for the Peterson model, the state vector of markings stored explicitly hardly reaches 16 bytes. Thus, we cannot expect our method to perform a reduction ratio better than $40\% - 60\%$. It appears that our method is not quite adapted for the storage of state spaces with small state vectors (< 20 bytes). That is due to the fact that the overhead needed to store a Δ-marking as well as extraneous data used to store markings, mainly pointers and additional bits added to vectors to fit in a machine word, are too important to enable a satisfactory reduction of the state space. Finally, the increase of the execution time remains acceptable for low values of k_δ but tends to grow with k_δ.

Experimentation on models extracted from programs. In a second step, we made experiments on two colored Petri nets automatically translated from concurrent Ada programs by the tool Quasar [24]. Both programs make use of advanced features of Ada tasking such as dynamic task creation. The first one is a client / server program with dynamic creation of servers to handle the requests of the clients. The second one is an Ada implementation of the sieves of Eratosthene to find all the prime numbers between 2 and N.

The results of our experimentations are reported in table 2. For both programs, we made four series of tests: without any compression technique, i.e., no state collapsing and no Δ encoding, (column *No comp.*), with the state collapsing

Table 2. Results obtained for programs

	No comp.	Collapse	Δ			Δ + Collapse		
			$k_\delta = 10$	$k_\delta = 20$	$k_\delta = 50$	$k_\delta = 10$	$k_\delta = 20$	$k_\delta = 50$
Client / server program								
N=4, 10 running tasks, 34 731 states								
M	9.45	1.37	1.89	1.42	1.15	0.36	0.30	0.26
T	00:00:02	00:00:03	00:00:03	00:00:03	00:00:06	00:00:03	00:00:04	00:00:07
V	285.17	41.26	56.93	42.71	34.61	11.01	9.10	8.00
N=5, 12 running tasks, 635 463 states								
M	205.63	28.37	36.80	21.94	20.96	6.84	4.98	4.85
T	00:00:51	00:01:54	00:01:11	00:01:44	00:02:54	00:01:18	00:01:52	00:03:04
V	339.31	46.82	60.73	36.20	34.59	11.29	8.21	8.00
N=6, 14 running tasks, 13 805 931 states								
M	-	684.41	1 035.15	962.489	447.73	176.90	167.85	105.33
T	-	00:26:04	00:00:00	00:44:20	01:36:33	00:38:43	00:48:59	01:38:17
V	-	51.98	78.62	73.10	34.01	13.44	12.75	8.00
Eratosthene program								
N=20, 9 running tasks, 3 599 634 states								
M	698.74	214.51	130.70	100.72	78.67	46.15	37.28	30.75
T	00:07:10	00:08:05	00:09:37	00:12:05	00:20:25	00:11:07	00:14:11	00:22:34
V	203.54	62.49	38.06	29.34	22.92	13.44	10.86	8.96
N=25, 10 running tasks, 24 884 738 states								
M	-	-	933.75	676.24	539.47	334.79	260.289	220.77
T	-	-	01:30:13	01:53:50	03:06:11	01:45:05	02:07:02	03:22:54
V	-	-	39.35	28.49	22.73	14.11	10.97	9.30
N=30, 11 running tasks, 96 566 610 states								
M	-	-	-	-	-	1 331.37	1 026.89	843.32
T	-	-	-	-	-	10:21:50	12:25:40	17:17:31
V	-	-	-	-	-	14.46	11.15	9.16

method enabled (column *Collapse*), with the Δ-markings method enabled (column Δ), and finally, with both methods enabled (column Δ + *Collapse*). For each run, row M reports the size in megabytes of the state space, row T reports the execution time of the search in the form hh:mm:ss, and row V reports the average size of the state vector in bytes. Some searches ran out of memory. This is indicated by a "-" in the table.

The reduction of the size of the state space for these models is much more impressive than for the simple models previously considered. Even for the lowest value of k_δ the reduction observed is significant. Without any compression technique enabled, the search is limited to state spaces with a few millions of states. State collapsing provides good results for both examples for a slight increase of the run time, but used without our method it is limited to state spaces with 10^7 states. For the Eratosthene program, our method clearly outperforms its competitor in terms of memory usage, even for the lowest value of k_δ. Finally, we observe that the best results are naturally obtained when combining both methods. For the more aggressive compression scheme (Δ + Collapse, $k_\delta = 50$), the search only consumed 2% to 5% of the space required without any state

compression, reducing the average size of the state vector to less than 10 bytes, whatever the model is. Such a drastic reduction comes without an unbearable cost: for the compression scheme mentioned, the execution time only tripled on all the examples.

The results observed confirm the analysis made in the previous section: for high values of k_δ the average size of the state vector is no more related to the model, and it tends to a limit which is the size of a collapsed vector (8 bytes in these examples).

6 Discussion and Perspectives

The two main techniques that are employed by model checkers to tackle the state explosion problem are partial order methods, e.g., persistent sets, and symmetry based reductions.

It is straight forward to see that the Δ-markings method is fully compatible with the first ones. Indeed, partial order methods are based on a selective search algorithm which only selects at each state a subset of the enabled transitions to generate the immediate successors of the state. It is therefore fully independent from the representation of the state space used.

Concerning symmetry reduction, the problem is more difficult. Computing symmetries often requires to permute threads or objects in the state vector. This means that the difference between two states is no more as simple as a single transition instance. Some other informations may thus also be saved in a Δ-marking to represent symmetries. Consequently, the algorithm could need more space, and also more time since the reconstitutions could be slowed. The problem of an efficient combination of the two methods is therefore an open problem that we are currently addressing.

When model checking problems of industrial sizes, it is preferable to not keep in memory all the visited states. Some techniques designed to achieve this have been presented in section 2. At first glance, it seems hard to combine our method with a "partial storage" method. This is so because Δ-markings point to some other states of the state space, disallowing, a priori, the possibility to not store every state in the reachability set. However, we see a possibility to combine our method with Geldenhuys's stratified caching [10]. The underlying idea of this caching policy is to systematically store and keep in the cache all the states met at given depths. Strata are thus classified as available and unavailable. States belonging to available strata may be replaced by new ones during the search, while the other ones must remain in the cache. The goal is to place an upper bound on the extra work realized by limiting the lengths of redundant exploration paths.

A solution to combine both methods is to allow the replacement of markings for which we are sure that no other Δ-marking points to it. When the first replacement occurs these markings are those who do not belong to the DFS stack and which have been met at depths $k_\delta - 1$, $2.k_\delta - 1$, ..., $n.k_\delta - 1$. Indeed all the successors of these markings are stored explicitly in the state space, or

point to another marking. Once all these states of this available strata have been replaced, the next available strata are those at depths $k_\delta - 2$, $2.k_\delta - 2$, ..., $n.k_\delta - 2$ and so on. By this way, we can guarantee that, at each step of the algorithm, each Δ-marking points to a marking which has not been replaced. Figure 4 illustrates our purpose. Parameter k_δ is set to 3. In the configuration depicted, markings at depth 2 have already been replaced by the algorithm. Markings at depth 1 are now eligible for replacement. Markings m and m' will become eligible as soon as they leave the stack.

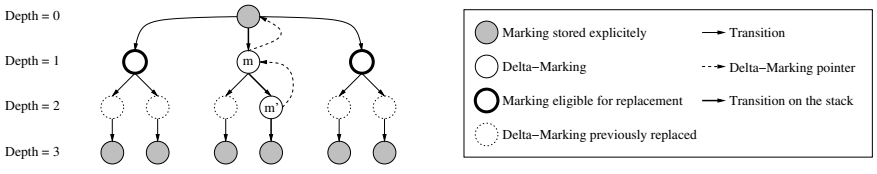


Fig. 4. Combining stratified caching and the Δ-markings method

7 Conclusions

We have presented in this work the Δ-markings method to store the state space of colored Petri nets. The basic idea of this method is to store a large set of states of the system in a non explicit way by only storing references on other states. Some optimizations which considerably reduce the run time penalty caused by the method have also been presented. In addition, our method can be combined with a state collapsing compression scheme to push the compression one step further. This simple idea reveals to be very useful in practice. The results of our experiments have shown the efficiency of our approach. On all examples, even for models issued from the translation of concurrent Ada programs, our experiments show that the average size of the state vector is close to 10 bytes. Furthermore, our technique does not increase the execution time in an unacceptable way.

Another appreciable property of our method is that it gives to the user the ability to specify the desired compression ratio, through the parameter k_δ.

References

1. Visser W. Memory efficient state storage in spin. In *SPIN'1996*, pages 21–35.
2. Holzmann G.J. State compression in spin : Recursive indexing and compression training runs. In *SPIN'1997*.
3. Geldenhuys J. and Valmari A. A nearly memory-optimal data structure for sets and mappings. In Thomas Ball and Sriram K. Rajamani, editors, *SPIN'2003*, volume 2648 of *LNCS*, pages 136–150. Springer.
4. Grègoire J.C. State space compression in spin with getss. In *SPIN'1996*, pages 90–108.
5. Parreaux B. Difference compression in spin. In *SPIN'1998*.

6. Geldenhuys J. and de Villiers P.J.A. Runtime efficient state compaction in spin. In Dennis Dams, Rob Gerth, Stefan Leue, and Mieke Massink, editors, *SPIN'1999*, volume 1680 of *LNCS*, pages 12–21. Springer.
7. Pastor E. and Cortadella J. Efficient encoding schemes for symbolic analysis of petri nets. In *Conference on Design, Automation and Test*, pages 790–795. IEEE Computer Society, 1998.
8. Schmidt K. Using petri net invariants in state space construction. In Hubert Garavel and John Hatcliff, editors, *TACAS'2003*, volume 2619 of *LNCS*, pages 473–488. Springer.
9. Godefroid P., Holzmann G.J., and Pirottin D. State-space caching revisited. In Gregor von Bochmann and David K. Probst, editors, *CAV'1992*, volume 663 of *LNCS*, pages 178–191. Springer.
10. Geldenhuys J. State caching reconsidered. In Susanne Graf and Laurent Mounier, editors, *SPIN'2004*, volume 2989 of *LNCS*, pages 23–38. Springer.
11. Parashkevov A.N. and Yantchev J. Space efficient reachability analysis through use of pseudo-root states. In Ed Brinksma, editor, *TACAS'1997*, volume 1217 of *LNCS*, pages 50–64. Springer.
12. Christensen S., Kristensen L.M., and Mailund T. A sweep-line method for state space exploration. In Tiziana Margaria and Wang Yi, editors, *TACAS'2001*, volume 2031 of *LNCS*, pages 450–464. Springer.
13. Mailund T. and Westergaard M. Obtaining memory-efficient reachability graph representations using the sweep-line method. In Kurt Jensen and Andreas Podelski, editors, *TACAS'2004*, volume 2988 of *LNCS*, pages 177–191. Springer.
14. Schmidt K. Automated generation of a progress measure for the sweep-line method. In Kurt Jensen and Andreas Podelski, editors, *TACAS'2004*, volume 2988 of *LNCS*, pages 192–204. Springer.
15. Behrmann G., Larsen K.G., and Pelánek R. To store or not to store. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *CAV'2003*, volume 2725 of *LNCS*, pages 433–445. Springer.
16. Godefroid P. Model checking for programming languages using verisoft. In *POPL'1997*, pages 174–186.
17. Holzmann G.J. *Design and validation of computer protocols*. Prentice Hall, 1991.
18. Leroy D. and Wolper P. Reliable hashing without collision detection. In Costas Courcoubetis, editor, *CAV'1993*, volume 697 of *LNCS*, pages 59–70. Springer.
19. Jensen K. Coloured petri nets: A high level language for system design and analysis. In Grzegorz Rozenberg, editor, *ATPN'1989*, volume 483 of *LNCS*, pages 342–416. Springer.
20. Dwyer M.B., Hatcliff J., Iosif R., and Robby. Space-reduction strategies for model checking dynamic software. *Electronic Notes in Theoretical Computer Science*, 89(3), 2003.
21. Lerda F. and Visser W. Addressing dynamic issues of program model checking. In Dragan Bosnacki and Stefan Leue, editors, *SPIN'2001*, volume 2057 of *LNCS*, pages 80–102. Springer.
22. Gerard J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
23. Evangelista S. High level petri nets analysis with helena. In Gianfranco Ciardo and Philippe Darondeau, editors, *ATPN'2005*, volume 3536 of *LNCS*, pages 455–464. Springer.
24. Evangelista S., Kaiser C., Pradat-Peyre J.F., and Rousseau P. Quasar : a new tool for analyzing concurrent programs. In Jean-Pierre Rosen and Alfred Strohmeier, editors, *Ada-Europe'2003*, volume 2655 of *LNCS*, pages 168–181. Springer.

Counterexample-Based Refinement for a Boundedness Test for CFSM Languages

Stefan Leue and Wei Wei

Department of Computer and Information Science,
University of Konstanz,
D-78457 Konstanz, Germany
{Stefan.Leue, wei}@inf.uni-konstanz.de

Abstract. In precursory work we suggested an abstraction-based highly scalable semi-test for the boundedness of Communicating Finite State Machine (CFSM) based modelling and programming languages. We illustrated its application to Promela and UML-RT models. The test is sound with respect to determining boundedness, but may return inconclusive "counterexamples" when boundedness cannot be established. In this paper we turn to the question how to effectively determine the spuriousness of these counterexamples, and how to refine the abstraction based on the analysis. We employ methods from program analysis and illustrate the application of our refinement method to a number of Promela examples.

1 Introduction

For various reasons the unboundedness of the maximal filling of communication channels in Communicating Finite State Machine (CFSM) [3] based modelling and programming languages at runtime is an undesirable property. First, unboundedness of communication buffers per se is an undesirable system feature since it points at a design flaw. Second, the unboundedness of a channel leads to an infinite state space and limits the applicability of finite state verification methods. Finally, in otherwise finite state models the unboundedness of a communication channel may lead to a model that reveals unpredictable behavior, for instance when a process attempts to write to a channel that has reached its specified capacity limit.

In precursory work [10,11] we have defined an incomplete semi-test for the boundedness of the message buffers in CFSM systems, which is an undecidable problem [3]. We have implemented the test in a tool named IBOC and applied it to CFSM models given in UML RT [10] and in Promela [11], the input language of the model checker SPIN [8]. We have also developed a method to estimate conservative upper bounds for the runtime filling of individual communication buffers in that work.

The boundedness semi-test that we devised is sound with respect to the boundedness of a given CFSM model. However, in case boundedness cannot be established, the test will return a verdict of "UNKNOWN". In that event the test also returns a "counterexample" which consists of a collection of control flow cycles in the state machines that collectively may lead to an unbounded flooding of at least one message buffer in the system. Our test is also based on a gross abstraction of the original CFSMs – we abstract from transition triggers, transition code, message orders, the activation conditions

P. Godefroid (Ed.): SPIN 2005, LNCS 3639, pp. 58–74, 2005.

of cycles, and cycle dependencies. As a consequence, the UNKNOWN verdict may be based on false negatives, i.e., the combinations of cycles that the test believes to be possible candidates for an unbounded flooding of message buffers may not be executable in the concrete model.

It is the objective of this paper to suggest methods that give the user automated support to a) determine, whether a counterexample is spurious or not, and b) to refine the abstraction in case a counterexample was found to be spurious. We will apply our method to models given in Promela, which is a modelling language that is very often used to model CFSM systems, in particular communication protocols. While Promela was a convenient choice, the ideas and concepts that we develop have wider applicability in the realm of concurrent, message based modeling and programming languages.

We employ methods from program analysis, in particular control flow and variable analysis to determine executability conditions for the cycles included in a counterexample. We have implemented our refinement method in the IBOC boundedness analysis tool and have applied our approach to various realistic and real-life Promela models.

Related Work. In our paper we follow the general idea of iterative counterexample guided abstraction refinement that was proposed in [4] and later applied to software model checking [1]. Since we use different code abstraction techniques the abstraction refinements that we propose are not comparable to the refinements proposed in those papers. In the context of linear programming based model checking the authors of [14] propose abstraction refinements based on the analysis of structural characteristics of control flow graphs. However, they do not address the imprecision that is caused by the abstraction of program code in concrete models. There is some similarity of our work to that on automating termination proofs using transition invariants [13,12,5,2]. The approaches presented in [12,5] only apply to program loops whose guarding conditions are no more complex than conjunctions of linear inequalities, whereas we can treat arbitrarily complex boolean conditions. The applicability of the approach in [2] to our problem is limited by its high complexity and the fact that its termination is not guaranteed.

Structure of the Paper. In Section 2 we review our previously published boundedness test and illustrate its application to Promela. In Section 3 we discuss the different sources of imprecision that lead to spurious counterexamples. Section 4 describes the static program analysis that we perform in order to abstract Promela code and illustrates how the abstraction can be refined in the presence of spurious counterexamples. We also employ graph structural criteria on the control flow cycle graphs to determine spuriousness and derive abstraction refinements, which we discuss in Section 5. In Section 6 we consider the complexities of the spuriousness detection and refinement method, and in Section 7 we present our experimental results.

2 Overview of the Boundedness Test

We now review the boundedness test for CFSM models first published in [10] and [11] and motivate the occurrence of spurious counterexamples. We review the test using Promela encodings of the CFSMs.

The objective of the boundedness test is to reduce the message passing behavior of a set of concurrent Promela processes to a number of control flow cycles. Each such cycle is then mapped to an effect vector that captures the number of messages sent minus the number of messages received for every message type in the system when executing that particular control flow loop. Linear inequality solving is then used on the resulting system of effect vectors to determine whether there is a linear combination of the vectors that leads to an unbounded "blow-up" of the number of messages of at least one message type in the system. If such a linear combination does not exist, we know that the system is bounded (test outcome BOUNDED). If a linear combination can be found, i.e., the linear inequality system has a solution, the system under consideration may be bounded or not (test outcome UNKNOWN). In this case, a counterexample is constructed from the particular solution to the linear inequality system.

```
mtype = {a, b};
chan ch1 = [1] of {mtype};
chan ch2 = [1] of {mtype};
active proctype Left(){
   do
   :: ch2 ? b -> ch1 ! a;
   od
}
```

```
active proctype Right(){
   byte x;
   x = 0;
   do
   :: (x == 0) -> ch2 ! b; x = 1;
   :: (x == 1) -> ch1 ? a; x = 0;
   od
}
```

Fig. 1. A simple Promela model

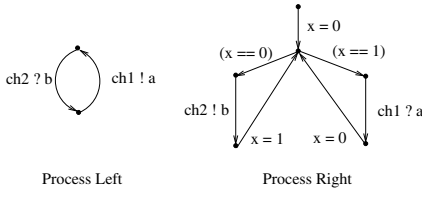


Fig. 2. The state machines of the Promela model in Figure 1

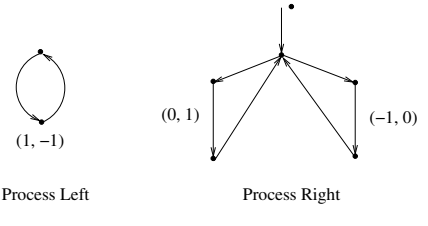


Fig. 3. The state machines with effect vectors of the Promela model in Figure 1

Consider the simple Promela model in Figure 1. The abstraction we perform in our test will first produce an extended CFSM model as in Figure 2. By ignoring variables and transition code this will be further abstracted into a system of state machines with effect vectors as illustrated in Figure 3. The three cycles entail the effect vectors (1, -1), (0, 1) and (-1, 0) which gives rise to the following system of integer linear inequalities: (1) $-x_1 + x_2 \geq 0$, (2) $x_1 - x_3 \geq 0$, and (3) $x_2 - x_3 > 0$. Note that the third equation is used to ensure that at least one component in the summary effect vector of each linear combination of cycle effect vectors attains a truly positive value.

The test will return a solution, in fact a linear combination with the values $x_1 = 0$, $x_2 = 1$, and $x_3 = 0$, and hence an outcome of UNKNOWN. Obviously, in the abstraction the left cycle of the process `Right` alone can cause the message buffer to blow up, and this is what the solution indicates. However, as it is easy to see, in the original Promela model an unbounded execution of this cycle without intervention of

the other cycles is not possible, which is ensured by the use of the (abstracted) program variable x.

Note that each cycle in the abstract system has a corresponding variable in the integer linear programming (ILP) problem into which the above system of inequalities is translated. This variable represents the number of executions of the respective cycle in a solution of the ILP problem if the ILP problem has a solution, i.e., if boundedness cannot be established. We call a set of cycles whose corresponding variables receive positive values in a solution to the ILP problem a *counterexample*. A counterexample represents a behavior of the system in which only the cycles in the counterexample are repeated infinitely often. Any other cycle in the system is either repeated only a finite number of times or not executed at all. A counterexample is said to be *spurious* if the behavior that it denotes is not a valid execution of the original model. In the above example, the test has found a spurious counterexample. It is the objective of the work in this paper to automatically detect spurious counterexamples generated by the boundedness test, and to refine the Promela model abstraction to exclude the spurious counterexamples that have been detected.

3 Sources of Imprecision

In this section we study the causes for the introduction of spurious counterexamples in our boundedness test. We also examine to what extent each cause affects the precision of the boundedness test.

3.1 Counterexamples and Spuriousness

The introduction of spurious counterexamples is a consequence of the conservative abstraction steps that we perform in the course of our boundedness test. We reconsider each of these abstraction steps to examine which information is removed from models during the step and how significant it affects the precision of the boundedness test. Note that these abstraction steps are conceptual and do not correspond to the concrete abstraction steps that the IBOC tool performs.

Step 1: Code Abstraction. In this step the program code in a model is abstracted away. The resulting CFSM system retains only the finite control structure and the message passing behavior of the model. We lose all the information about how the behavior of the model is constrained by the conditions on variables that are imposed by the program code. Losing such information is very significant because it often depends on the runtime value of a variable whether to send or receive a message, which message to send or receive, where messages are to be sent or from where messages are to be received. We will therefore consider this source of imprecision in more detail.

Step 2: Abstraction from Message Orders. In this step we neglect all information regarding the order of messages in message buffers. In particular, we assume that a message is always available to trigger a transition wherever it is in the buffer. This can be too coarse an overapproximation for a model that employs strict first-in-first-out message

buffers. However, models in practice usually have a message deferral/recall mechanism that stores an arriving message which cannot immediately be processed by the system into a special buffer so that it can be recalled when it is later needed. This is consistent with the semantics of our abstraction. In other words, this abstraction step does not introduce imprecision in most practical situations and we will therefore not address it in this paper.

Step 3: Abstraction from Activation Conditions. In this step the activation conditions of control flow cycles are abstracted away. We assume that there are always enough messages of the right type available for a cycle to be reachable from the initial configuration of the model. This assumption is reasonable in practice since an unreachable cycle has no influence on the system behavior and usually indicates a design error. We will therefore not consider this source of imprecision either.

Step 4: Abstraction from Cycle Dependencies. In this step we abstract from implicit dependencies between control flow cycles. We consider different types of such dependencies: exclusion dependencies or inclusion dependencies, global dependencies or local dependencies. An exclusion dependency forbids a set of cycles to be jointly repeated infinitely often, while an inclusion dependency stipulates the need of some cycles being repeated infinitely often to enable other cycles to be repeated infinitely often. A global dependency specifies a dependency among cycles in multiple processes, while a local dependency relates cycles in one process. Cycle dependencies are imposed either by the program code executed while the program goes through a cycle, or by structural characteristics of the control flow graphs. Disregarding cycle dependencies means that arbitrary cyclic executions can be combined to form a potentially spurious counterexample, which is why we will further address this source of imprecision.

3.2 Boolean Conditions on Cycles

We now consider the impact of the abstraction of boolean executability conditions during code abstraction. As an example, consider the cycle shown in Figure 4[1]. The cycle guard checks whether the runtime value of a local integer variable i is less than 4. If the condition is satisfied, then a message `msg` is sent to the channel `ch`, and i is incremented. After the code abstraction, the resulting abstract cycle in Figure 5 retains only the message sending statement, and all the information about i has been removed. The abstract cycle will inevitably by itself form a counterexample in which a message `msg` is sent to `ch` without any constraint each time the cycle is executed. Apparently, this counterexample is spurious since the cycle can be repeated without interruption at most 4 times, if i is initialized to 0. If we want the cycle to be repeated later, then some other cycle in the control flow graph of the same process in which the value of i will be changed back to be less than 4 needs to be executed.

Figure 6 shows such a cycle in which the value of i is reset to 0. If this cycle is the only other cycle that modifies i, then the boolean condition on the cycle in Figure 4 imposes an *inclusion* dependency between these two cycles. In other words, if the cycle

[1] All program code in examples will be given in Promela syntax.

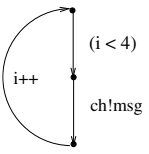


Fig. 4. A simple cycle

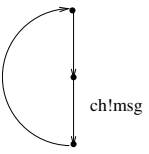


Fig. 5. The abstract cycle

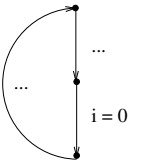


Fig. 6. A cycle resetting i

in Figure 4 appears in a counterexample, then the cycle in Figure 6 must be included in the same counterexample.

3.3 Graph-Structural Dependencies

Graph-structural dependencies are another source of cycle dependencies that may help to reveal spurious counterexamples. Consider two cycles in the control flow graph of one same process that reside in two different strongly connected components. Since at least one cycle is not reachable from the other, they cannot be jointly repeated infinitely often. Strongly connected components induce *exclusion* dependencies.

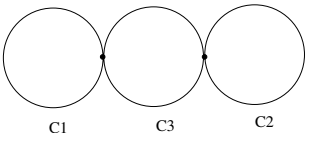


Fig. 7. Three cycles

There is another kind of graph-structural cycle dependencies. Consider the control flow graph of a process as shown in Figure 7. Assume that a counterexample contains the cycles `C1` and `C2` and does not contain the cycle `C3`. Note that `C1` and `C2` do not share a common state, which implies that it is impossible to repeat `C1` and `C2` infinitely often without repeating `C3` infinitely often. Since `C3` is not included in the above counterexample it is spurious. The resulting dependency is an *inclusion* dependency.

Summary. We recognize the following two types of information as being crucial to the precision of our unboundedness test: (1) cycle dependencies imposed by the boolean conditions on cycles, and (2) cycle dependencies imposed by structural characteristics of control flow graphs. In the next two sections, we will propose several automated refinement methods based on counterexample spuriousness analyses with respect to these two types of information.

4 Cycle Code Analysis

It is generally impossible to precisely determine cycle inclusion dependencies. Our method is therefore incomplete and it overapproximates the actual inclusion dependencies. As a consequence, (1) a counterexample can still be spurious even if it does

not violate the determined dependency, and (2) some spurious counterexamples may never be excluded.

A cycle in a control flow graph corresponds to a loop in the program code of the respective process. The task of a loop is either to continuously react to stimuli from the environment, or to perform some local information processing task. As mentioned above, we also distinguish global and local dependencies. Local dependencies are determined by cycle conditions in which all the variables are locally modified. This excludes the use of local variables to store the contents of messages since this would imply global dependencies. In this paper we will focus solely on local dependencies and leave the treatment of global dependencies for future work.

Our approximative method to determine inclusion dependencies is only applicable to loop code that follows a certain syntactic pattern:

- There exists at least one branch statement or loop statement in the loop with a guarding boolean expression in which all the variables are only locally modified within the loop. We call a variable occurring in the guard of a branch statement or a loop statement a *control variable*.
- The computation on control variables involves only linear expressions over integers, such as an incrementation or a decrementation. The guards of branch statements and loop statements are boolean conditions that involve only comparisons of linear expressions over control variables.

We notice that this is not overly constraining since code following this pattern is commonly used in real models. We further assume that all the program statements are side-effect free, and that all function calls have been inlined.

4.1 Constraints on Repeated Cycle Executions

A condition statement in a cycle functions as the guard on the executability of a statement within a loop. Since program code can have nested loops, a cycle can have several condition statements that guard the loop statements at different depths. We study how the repeated executions of a cycle are constrained by each of the condition statements in the cycle.

We denote a condition statement by a boolean expression enclosed in a pair of parentheses. Consider a condition statement `(`B`)` in a cycle `C1`. `(`B`)` is executable if and only if B evaluates to `true` under the current valuation of variables. We are interested in determining max_B, the maximal number of times that `(`B`)` can be repeatedly executed if all variables in B are only modified within `C1`. If max_B exists, then we can draw the following conclusions: (1) `C1` cannot be consecutively repeated more than max_B times. (2) For every max_B times that `C1` is repeated, at least one of the *neighboring* cycles of `C1` has to be executed. A cycle is a neighboring cycle of another cycle if they share common states. (3) For every max_B times that `C1` is repeated, at least one of the *supplementary* cycles with respect to B has to be executed. A supplementary cycle of `C1` with respect to B is a cycle that modifies at least one of the variables in B in a way that renders B satisfied. The conclusions (2) and (3) impose an inclusion cycle dependency that can be used to determine counterexample spuriousness and to refine the abstract system of the original model. However, it is generally impossible to

precisely determine max_B and we will hence resort to computing an upper bound for max_B. It is easy to see that this is a safe approximation. For any number n, the restriction that other cycles have to be executed every n times that `C1` is executed does not exclude the possibility that other cycles are executed every m times that `C1` is repeated for any $m < n$.

We notice that a boolean expression can be converted into its negation free disjunctive normal form, where each disjunct is a conjunction of one or more positive atomic propositions. Because the boolean conditions that we consider involve only comparisons of linear expressions, an atomic proposition is a linear (in)equality. In the sequel we assume that any boolean expression is given in this type of normal form.

For each disjunct d of B, we denote by $max_{B,d}$ the maximal number of times that d is satisfied when `(`B`)` is repeatedly executed without the variables in B being modified outside `C1`. If we can determine $max_{B,d}$ values, then max_B is bounded by the sum of all the $max_{B,d}$ values. Note that max_B can be larger than any $max_{B,d}$ value because it is not always the case that two disjuncts d_1 and d_2 of B are satisfied at the same time. For each (in)equality l of d, we denote by $max_{B,d,l}$ the maximal number of times that l is satisfied when `(`B`)` is repeatedly executed without the variables in B being modified outside `C1`. We can further reduce the computation of $max_{B,d}$ to the computations of $max_{B,d,l}$ values. $max_{B,d}$ is the minimum of all the $max_{B,d,l}$ values, because if any (in)equality l of d is not satisfied then d is not satisfied.

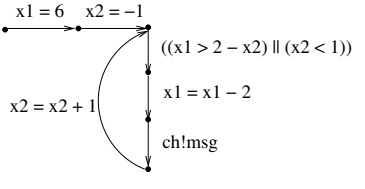


Fig. 8. Cycle `C`

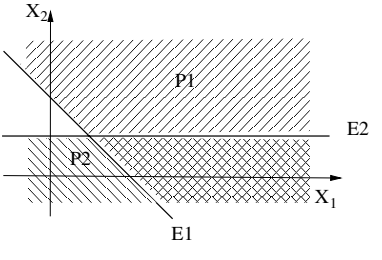


Fig. 9. Convex polyhedra

Consider as an example the cycle `C` in Figure 8. x_1 and x_2 are integer variables. Before `C` is entered, x_1 is initialized with 6 and x_2 is initialized with -1. During each cycle execution, the value of x_1 is decremented by 2 and the value of x_2 is incremented by 1. Let $B = (x_1 > 2 - x_2) \vee (x_2 < 1)$, $d_1 = l_1 = (x_1 > 2 - x_2)$, and $d_2 = l_2 = (x_2 < 1)$. It can be manually determined that $max_{B,d_1,l_1} = 3$, $max_{B,d_1} = 3$, $max_{B,d_2,l_2} = 2$, $max_{B,d_2} = 2$, and $max_B = 3$. An upper bound of max_B is 5 (3+2), which is larger than 3 because d_1 and d_2 can be both satisfied at same time during cycle executions. We conservatively take 5 to overapproximate the actual max_B value 3.

Before we explain the method to compute $max_{B,d,l}$, we give a geometric illustration of the problem. In the example from Figure 8 each of B's disjuncts is a system of linear (in)equalities that defines a convex polyhedron in the 2-dimensional Euclidean space as shown in Figure 9. The polyhedron `P1` is defined by the disjunct d_1 and the polyhedron `P2` is defined by the disjunct d_2. B is the union of all the polyhedra

defined by B's disjuncts. The (in)equalities of a disjunct define the edges of the polyhedron defined by the disjunct. For instance, the edge E1 is defined by the inequality $l_1 = (x_1 > 2 - x_2)$. E1 can be represented by the equation $x_1 + x_2 = 2$, in which no variables occur in the right-hand side. We call $x_1 + x_2$ the *control expression* of l_1, and 2 the *boundary value* of l_1.

Let (x_1, x_2) denote the point defined by x_1 and x_2 in the Euclidean space. During repeated executions of C, (x_1, x_2) moves in the space while x_1 and x_2 are modified. For a disjunct d of B, $max_{B,d}$ is the same value as the maximal number of computation steps that (x_1, x_2) can stay in the polyhedron defined by d. $max_{B,d}$ is therefore bounded if and only if the following *exiting property* is satisfied: at some point of time (x_1, x_2) will exit the polyhedron and never return. In the sequel we show a sufficient but not necessary condition of the exiting property.

During repeated execution of C we obtain a sequence of vectors (x_1^0, x_2^0),(x_1^1, x_2^1), ... that denotes the values of x_1 and x_2 at the end of each execution of C. Given an edge E of a polyhedron P, we assume that E is defined by an (in)equality l. Let $ce(l)$ denote the control expression of l and $bv(l)$ denote the boundary value of l. Let $ce(l)^i$ denote the value of $ce(l)$ when $x_1 = x_1^i$ and $x_2 = x_2^i$, where i is a nonnegative integer. Assume that the sequence $ce(l)^0$, $ce(l)^1$, ... is either strictly increasing or strictly decreasing, i.e., $ce(l)$ is modified monotonically. Let $d^i = |ce(l)^i - bv(l)|$ denote the distance between $ce(l)^i$ and the boundary value of l. If the sequence d^0, d^1, ... is strictly decreasing, then the point (x_1, x_2) is always moving closer to the edge E. Furthermore, the sequence d^0, d^1, ... must be finite. The sequence (x_1^0, x_2^0), (x_1^1, x_2^1), ... is then also finite, which implies that (x_1, x_2) can only stay in P for a finite number of computation steps. Note that max_{B,d_1,l_1} is identical to the length of the sequence of distances d^i. We conclude that under the above assumption the exiting property holds for P and we call E an *exit border* of P. Obviously, both E1 and E2 are exit borders of the respective polyhedron. The existence of an exit border is a sufficient condition for the exiting property to hold.

4.2 Computing $max_{B,d,l}$ Values

Given a cycle C1, let (*B*) denote one of the condition statements in C1, d a disjunct of B and l an (in)equality of d. We now address the question how to compute $max_{B,d,l}$.

Remember that l defines an edge E of the polyhedron that d defines. E can be represented as the equation $ce(l) = bv(l)$. In case that $ce(l) = bv(l)$ is an exit border, an upper bound of the maximal number of computation steps that are needed for the value of $ce(l)$ to reach $bv(l)$ can be determined, (1) if we can determine the possible differences of the runtime values of $ce(l)$, i.e., the step values of $ce(l)$, before and after each execution of C1, and (2) if we can determine the initial values of $ce(l)$ before C1 is entered and therefore the longest distance to the boundary value. We determine whether $ce(l) = bv(l)$ is an exit border by determining and analyzing the step values of $ce(l)$.

Determining Step Values. The cycle code imposes a relation constraining the values of variables before and after an execution of the cycle. Determining bounds for the step values of the control expression $ce(l)$ can be seen as an optimization problem. We generate a set of ILP problems from the cycle code, whose constraints reflect how the

runtime values of variables are changed and constrained by the cycle code. The objective functions of these ILP problems are either the maximum or the minimum of the step values of $ce(l)$. Note that both the maximum and the minimum of the step values must be computed in order to determine whether $ce(l)$ is modified monotonically. The generation of the constraints of an ILP problem for determining step values involves the transformation of each program statement in the cycle code into one or more (in)equations. Since not all the statements in the cycle code are relevant to the computation on the control variables in l, we compute a slice [16] of the cycle code with the slice criterion being $< (B), \bar{x} >$, where $\bar{x}$ are all the variables in l. Since we have the assumption that all the program statements are side-effect free, the transformation of program statements is straightforward. As an example, the assignment statement `x := x + 1` is transformed to the equation $x_{i+1} = x_i + 1$. The variable x_i denotes the runtime value of the variable x before the execution of the assignment, and the variable x_{i+1} denotes the runtime value of x after the execution. When a receive statement is executed, a variable x in the statement receives a random value that depends on the content of the received message. In the corresponding equation, we must represent the new value of x by a newly introduced variable. Moreover, when an array member is affected by an assignment statement or a receive statement, we take the conservative assumption that any member of the array may be affected.

```
max: x1_1 + x2_1 - x1_0 - x2_0;          min: x1_1 + x2_1 - x1_0 - x2_0;

x1_0 > 2 - x2_0;                         x1_0 > 2 - x2_0;
x1_1 = x1_0 - 2;                         x1_1 = x1_0 - 2;
x2_1 = x2_0 + 1;                         x2_1 = x2_0 + 1;


max: x1_1 + x2_1 - x1_0 - x2_0;          min: x1_1 + x2_1 - x1_0 - x2_0;

x2_0 < 1;                                x2_0 < 1;
x1_1 = x1_0 - 2;                         x1_1 = x1_0 - 2;
x2_1 = x2_0 + 1;                         x2_1 = x2_0 + 1;
```

Fig. 10. The generated integer programming problems

Consider the example in Figure 8. The cycle code is abstracted into the four ILP problems in Figure 10 for determining step values for the control expression $x_1 + x_2$ of the inequality $x_1 > 2 - x_2$. The top two ILP problems are used to determine the bounds on step values under the condition $(x_1 > 2 - x_2)$ as the first disjunct of the guard $(x_1 > 2 - x_2) \vee (x_2 < 1)$, while the other two ILP problems are used to determine the bounds under the condition $(x_2 < 1)$ as the second disjunct of the same guard.

Analyzing Step Values. The solutions to the objective functions of the ILP problems for determining the step values of $ce(l)$ set an upper bound and a lower bound on the actual step values. If the two bounds have different signs, then the value of $ce(l)$ may not be modified monotonically. If one of the two determined bounds is 0, then the value of $ce(l)$ may be unchanged forever. In these two cases, we fail to determine whether $ce(l) = bv(l)$ is an exit border, and conservatively set $max_{B,d,l}$ to be unbounded. If the upper bound and the lower bound are both positive, then the value of $ce(l)$ is always

increased. We conservatively take the lower bound as the constant step value of $ce(l)$ that denotes the slowest move of $ce(l)$. Similarly, if the upper bound and the lower bound are both negative, then we take the upper bound to be the constant step value. For the ILP problems in Figure 10, all the solutions are -1, which is then both the upper bound and the lower bound determined for the step values of the control expression $x_1 + x_2$. The constant step value determined for $x_1 + x_2$ is then -1.

Let $op(l)$ denote the *comparison operator* in l. We determine whether $ce(l) = bv(l)$ is an exit border according to $op(l)$ and the determined constant step value of $ce(l)$. If $op(l)$ is $=$, then $ce(l) = bv(l)$ is an exit border whatever the constant step value is, because any nonzero step value annuls the satisfaction of l. In this case we directly set $max_{B,d,l}$ to 1. Let $\bar{x}$ be the vector of the variables in B. If the constant step value of $ce(l)$ is positive, and if op is $<$ or $\leq$, then in the polyhedron defined by d the point defined by $\bar{x}$ is moving closer to the edge $ce(l) = bv(l)$. $ce(l) = bv(l)$ is then an exit border. Similarly, if the constant step value is negative, then $ce(l) = bv(l)$ is an exit border when op is $>$ or $\geq$. In all other cases, $ce(l) = bv(l)$ is not an exit border and we set $max_{b,d,l}$ to be unbounded. In the example in Figure 8, we have determined the constant step value of $x_1 + x_2$ to be -1. The comparison operator of the corresponding inequality is $>$. The edge $x_1 + x_2 = 2$ is therefore an exit border.

Determining Initial Values. If we determine $ce(l) = bv(l)$ to be an exit border, and if $op(l)$ is not $=$, then we need to determine the initial values of $ce(l)$ in order to know the longest distance from the initial values of $ce(l)$ to the boundary value. We use a simple solution that employs backward depth first searches to check each acyclic path leading to one of the entry locations of `C1` whether $ce(l)$ receives a constant value on that path before `C1` is entered. Our solution to determining initial values is certainly incomplete, but also more efficient than other approaches, such as interval analysis [6]. This is because our solution does not require the analysis of the whole control flow graph of the respective process. An interval analysis will cost even more when any variable in $ce(l)$ is affected somewhere outside `C1` by a received message sent from some other process. The sending process of that message has then to be analyzed even if the receiving of the message will not affect the initial value of $ce(l)$. On the other hand, we will show later that, even in the absence of the determined initial values of $ce(l)$, we may still be able to determine an inclusion dependency.

Computing $max_{B,d,l}$. Let $step$ denote the determined constant step value of $ce(l)$ and $distance$ the longest distance between $ce(l)$ and $bv(l)$. $\lceil \cdot \rceil$ is the ceiling function that returns the smallest integer greater than the input real number. $\lceil distance/step \rceil$ sets an upper bound on $max_{B,d,l}$ that is used to safely approximate the actual $max_{B,d,l}$ value. In the example in Figure 8, we determine $max_{(x_1>2-x_2)\vee(x2<1),x_1>2-x_2,x_1>2-x_2}$ as follows. The initial value of $x_1 + x_2$ is 5. $step = -1$ and $distance = 2 - 5 = -3$. The determined upper bound is then 3 $(-3 \div -1)$.

4.3 Abstraction Refinement

For a condition statement `(`B`)` in a cycle `C1`, if we can determine an approximative max_B value, then we know that one of the neighboring cycles of `C1` and one of the

supplementary cycles of C1 with respect to B have to be executed every max_B times C1 is executed. A counterexample containing C1 is determined to be spurious, if (1) no neighboring cycle of C1 is included in the counterexample, or if (2) no supplementary cycle of C1 with respect to B is included in the counterexample.

However, it is generally impossible to determine the exact set of supplementary cycles with respect to B, since it is generally impossible to determine whether the execution of a cycle has the effect to render B satisfied. A simple overapproximation is to consider a cycle a supplementary cycle if it resides in the same process as C1 and if it modifies at least one of the variables in B.

Refinement with max_B. We use max_B, the set of neighboring cycles, and the set of supplementary cycles to refine the abstract system of the original model. We first add to the boundedness test ILP problem one additional constraint which enforces that every max_B times that C1 is executed least one of its neighbors will be executed. Assume that the set of neighboring cycles of C1 is NC. The added constraint is: $c_1 \leq max_B \times \sum_{C_i \in NC} c_i$, where c_1 is the variable corresponding to C1 and c_i is the variable corresponding to $\mathtt{C}_i$. A similar constraint is added to the boundedness test ILP problem, which requires at least one supplementary cycle to be executed each max_B times that C1 is executed.

Refinement without max_B. Let $\bar{x}$ denote the vector of the variables in B. If all disjuncts of B contain an (in)equality that defines an exit border, then we know that the number of computation steps for which the point defined by $\bar{x}$ can stay in the polyhedron of each disjunct is always finite. In this case, even if max_B cannot be determined, we can still determine an inclusion dependency. After a finite but unknown number of times C1 is executed, C1 will be exited, and one of its neighboring cycles as well as one of its supplementary cycles have to be executed. Let SC be the set of supplementary cycles of C1 with respect to B. In the refinement without max_B, the boundedness test ILP problem is replaced by two ILP problems, each augmenting the original ILP problem with one of the following constraints: (1) $c_1 = 0$, or (2) $c_1 > 0 \wedge \sum_{C_i \in NC} c_i > 0 \wedge \sum_{C_i \in SC} c_i > 0$, where c_1 is the variable corresponding to C1 and c_i is the variable corresponding to $\mathtt{C}_i$. These two constraints stipulate that (1) either C1 is not repeated infinitely often, or (2) at least one of the neighboring cycles and at least one of the supplementary cycles have to be repeated infinitely often. The two newly generated ILP problems partition the behavior of the refined abstract system into two disjoint subsets. The original model can be determined to be bounded if both of the two ILP problems are infeasible.

During experiments we found that this approximative way of determining supplementary cycles is rather coarse for some models in that it prevents many spurious counterexamples from being excluded. Consider the cycle in Figure 8. After the condition $(x_1 > 2 - x_2) \vee (x_2 < 1)$ is not satisfied and C is exited, assume that some supplementary cycles are going to be executed to make the condition satisfied again. Assume a neighboring cycle C′ of C, in which no message is sent or received. If each execution of C′ decreases the value of x_1 and increases the value of x_2, then C′ has the same effect on x_1 and x_2 as C does. In particular, it cannot render $(x_1 > 2 - x_2) \vee (x_2 < 1)$ satisfied. However, C′ is regarded a supplementary cycle of C. The refinement of the

abstract system with the determined set of supplementary cycles including C' does not exclude the spurious counterexample consisting of C and C'.

We adopt a finer solution if, for every (in)equality l in B, $ce(l)$ is modified monotonically within C1. In this case, we regard a cycle C2 a supplementary cycle with respect to B, if C2 satisfies the following condition. There exists an (in)equality l in B that defines an exit border. The value of $ce(l)$ is increased within C2 if it is decreased within C1. The value of $ce(l)$ is decreased within C2 if it is increased within C1. This solution is more expensive since it involves code analysis for each cycle that modifies one or more variables in B, but is more precise in determining spuriousness.

5 Graph-Structural Dependency Analysis

In this section we consider two types of cycle dependencies, namely those imposed by strongly connected components and those imposed by direct connectedness of cycles, and propose appropriate abstraction refinements.

Given a counterexample in which two cycles in one same process do not share common states, some other cycles in the same process have to be included in the counterexample to "bridge" them. We introduce the concept of *self-connected cycle set*. A set of cycles is self-connected if any two cycles in the set are reachable from each other by traversing through only the cycles in the set. A counterexample is spurious if, for some process P, the set of all the cycles of P in the counterexample is not self-connected.

We propose the following refinement. Given a counterexample, if there are two cycles C1 and C2 of one same process P in the counterexample that are not reachable from each other by traversing through only the cycles in the counterexample, then we determine all the self-connected sets of cycles of P that contain both C1 and C2. If no such set exists, then C1 and C2 are in different strongly connected components. Let c_1 be the variable corresponding to C1 and c_2 be the variable corresponding to C2. We replace the boundedness test ILP problem with three ILP problems, each of which augments the original ILP problem with one of the following three constraints: (1) $c_1 = 0 \wedge c_2 = 0$; (2) $c_1 = 0 \wedge c_2 > 0$; (3) $c_1 > 0 \wedge c_2 = 0$. These three constraints prevent C1 and C2 from being both repeated infinitely often.

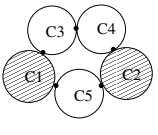


Fig. 11. A control-flow graph

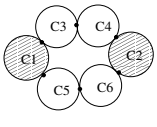


Fig. 12. A control-flow graph

If there exists at least one self-connected set of cycles containing both C1 and C2, then C1 and C2 are in the same strongly connected component. Let n denote the number of determined self-connected cycle sets. To refine the abstract system, we generate $n+3$ new ILP problems to replace the original boundedness test ILP problem. Three of them are the same ILP problems as generated in case C1 and C2 belong to different strongly connected components. The other n ILP problems stipulate that, if C1 and C2 are both

repeated infinitely often, then all the cycles in one of the determined self-connected cycle sets have to be repeated infinitely often. For each determined self-connected cycle set S_i ($1 \leq i \leq n$), a new ILP problem is generated by augmenting the original ILP problem with the following constraint: $c_1 > 0 \wedge c_2 > 0 \wedge \bigwedge_{C \in S_i} c > 0$, where c is the variable corresponding to the cycle C. As an example, we assume that the control-flow graph of P is the same as shown in Figure 11. All the self-connected sets of cycles containing C1 and C2 are $S_1 = \{\text{C1, C3, C4, C2}\}$ and $S_2 = \{\text{C1, C5, C2}\}$. Let c_i ($3 \leq i \leq 5$) denote the variable corresponding to the cycle Ci. Each of the 5 newly generated ILP problems has one of the following constraints: (1)$c_1 = 0 \wedge c_2 = 0$; (2) $c_1 = 0 \wedge c_2 > 0$; (3) $c_1 > 0 \wedge c_2 = 0$; (4) $c_1 > 0 \wedge c_2 > 0 \wedge c_3 > 0 \wedge c_4 > 0$; (5) $c_1 > 0 \wedge c_2 > 0 \wedge c_5 > 0$.

A caveat of this method is that the number of newly generated ILP problems can be exponential in the number of the cycles of P. We propose an alternative method that only generates 4 ILP problems each time the abstract system is refined. Consider the control-flow graph in Figure 11. To reach C2 from C1, one of the neighboring cycles of C1 must be entered. Similarly, one of the neighboring cycles of C2 must be entered in order to reach C1 from C2. Instead of generating the two ILP problems corresponding to the two self-connected cycle sets containing C1 and C2, we can build only one ILP problem by augmenting the original boundedness test ILP problem with the following constraint: $c_1 > 0 \wedge c_2 > 0 \wedge c_3 + c_5 > 0 \wedge c_4 + c_5 > 0$. To generalize this idea we assume that two cycles C and C′ in a counterexample are not reachable from each other by traversing only the cycles in the counterexample. We compute a sequence of pairs of cycle sets: $< N_C^0, N_{C'}^0 >, < N_C^1, N_{C'}^1 >, ..., < N_C^n, N_{C'}^n >$. N_C^0 is the set of neighboring cycles of C. N_C^{i+1} contains all the cycles that neighbor some cycle in N_C^i and that are not in any N_C^j if $j \leq i$. $N_{C'}^i$'s ($0 \leq i \leq n$) are defined likewise. The computation of the sequence $< N_C^0, N_{C'}^0 >, ..., < N_C^n, N_{C'}^n >$ terminates if, for some number n, either (1) N_C^n or $N_{C'}^n$ is empty, or (2) there is a cycle in N_C^n and a cycle in $N_{C'}^n$ such that these two cycles are neighbors. If the first condition is true, then C and C′ must be in different strongly connected components. The treatment in this case is the same as in the previous refinement method. If the second condition is true, then we know that, to reach C′ from C, one cycle from each set in the sequence $N_C^0, N_C^1, ..., N_C^n, N_{C'}^n, N_{C'}^{n-1}, ..., N_{C'}^0$ has to be entered. A similar requirement is for reaching C from C′. To refine the abstract system, we replace the boundedness test ILP problem with 4 newly generated ILP problems. Three of them augment the original ILP problems with the constraints that prevent C and C′ from being both repeated infinitely often. The last ILP problem augments the original ILP problem with the constraint which stipulates that, if C and C′ are both repeated infinitely often, then at least one cycle from each set in the sequence $N_C^0, N_C^n, ..., N_{C'}^1, ..., N_{C'}^n$ has to be also repeated infinitely often. Note that, although this method generates fewer ILP problems, it is coarser in that some spurious counterexample may not be excluded. Consider the control flow graph in Figure 12. Let c_i ($1 \leq i \leq 6$) be the variable corresponding to the cycle Ci. If a counterexample contains C1 and C2 and does not contain any of other cycles in the same control flow graph, then a constraint is added to one of the newly generated ILP problems as following: $c_1 > 0 \wedge c_2 > 0 \wedge c_3 + c_5 > 0 \wedge c_4 + c_6 > 0$. This constraint

cannot exclude a potentially spurious counterexample that contains `C1`. `C2`, `C3`, `C6`, and not others in the same control flow graph.

6 Complexity

It is an undecidable problem to determine whether a counterexample generated in the boundedness test is spurious or not. The proposed refinement method in this paper is incomplete, and inevitably has a high theoretical complexity due to the following facts. (1) The number of generated ILP problems for determining the step values of a control expression is exponential both in the size of the respective cycle and in the size of the guard boolean expression of each condition statement in the cycle. (2) Solving an ILP problem is NP-complete.

Despite a high theoretical complexity, the proposed refinement method is efficient in practice. The reasons are the following. (1) A cycle usually has a small portion relevant to the computation of the control variables. (2) A guard boolean expression of a condition statement is usually not complex such that its negation free disjunctive normal form contains only a small number of disjuncts. (3) A generated ILP problem for determining step values is usually very small.

7 Experimental Results

We implemented the spuriousness determination and counterexample-based refinement methods that we propose in the IBOC system . We report the experimental results of analyzing the following models using IBOC on a Pentium IV 3.20GHz machine with 2GB memory: the *Sort* model included in the SPIN [8] 4.22 distribution package, a model of the *General Inter-ORB Protocol* (GIOP) [9], and a model of the *Model View and Concurrency Control Protocol* (MVCC) [15].

The *sort* model consists of 8 running processes that collaboratively sort 7 numbers by exchanging messages through 7 buffers. IBOC used 0.718 second to report a counterexample. The counterexample consists of only one cycle from the `Left` process that sends one of the 7 numbers to be sorted to the buffer `q[0]` in their initial order each time the cycle is executed. The cycle is guarded by the condition $counter < 7$ in which the variable `counter` receives the value 0 before the cycle is entered. Each execution of the cycle increments `counter` by 1. IBOC used another 0.688 second to determine that the cycle cannot be repeated consecutively more than 7 times, and that the cycle has neither neighboring cycles nor supplementary cycles to modify `counter`. IBOC therefore determined the counterexample to be spurious, and excluded the cycle from the abstract system. IBOC found no more counterexamples. The *sort* model was determined to be bounded.

The GIOP protocol supports message exchange and server object migration between object request brokers (ORBs) in the CORBA architecture. The Promela implementation [9] that we took is a real life model with considerable size and complexity. IBOC reported 5 counterexamples within 14.015 seconds. The first counterexample was determined to be spurious. IBOC failed to determine spuriousness for the second, the third, and the fourth counterexample because there is a cycle in each of them guarded

by a boolean condition that involves a variable used to store the content of received messages. These conditions induce global cycle dependencies, which the proposed refinement method in this paper cannot handle. The last reported counterexample was manually determined to be spurious. The reason that IBOC failed on it is the following. When IBOC determines a cycle to be supplementary to an analyzed cycle, it does not consider how many times the supplementary cycle has to be executed in order to enable the analyzed cycle. However, considering such information may lead to an exponential number of new boundedness test ILP problems to be generated with respect to the number of supplementary cycles. Each of the ILP problems corresponds to one potential combinatory effect of supplementary cycles.

The MVCC protocol is one of the underlying protocols of the *Clock* toolkit [7] for the development of groupware applications. It supports multi-user server-client communication and the synchronization of concurrent updates of information. We took the Promela implementation in Appendix A of [15] that allows 2 clients. The model consists of 8 concurrently running processes. IBOC visited 70 states and 83 transitions in the state machines of the processes, and constructed 46 simple cycles and identified 16 types of messages. Within 4.281 seconds, IBOC found 4 counterexamples and determined the first 3 of them to be spurious. The last counterexample contains only one cycle in a `User` process (as a client) that sends a message without any constraints. It is a real counterexample.

8 Conclusion

In this paper we have presented abstraction refinement techniques for an incomplete communication channel boundedness test for Promela. Our approach allows one a) to determine spuriousness of counterexamples, and b) to refine the previous abstraction in order to exclude these spurious counterexamples. In order to determine spuriousness, we statically compute executability conditions for cycles and we analyse the cycles in the control flow graphs of the system to determine inclusion and exclusion dependencies. We implemented our spuriousness determination and the refinement method in IBOC. We presented experimental results that show that our method scales to systems of realistic size and is capable of returning meaningful results.

Further work includes the analysis of global cycle dependencies - the analysis of the GIOP protocol showed that a number of spurious counterexamples can not yet be detected due to the unavailability of this global analysis. A further goal is to apply our method to UML RT and UML 2.0 models that include state machine transitons attributed with Java code.

Acknowledgements. We thank Richard Mayr for initial discussions on the subject of this paper.

References

1. T. Ball, A. Podelski, and S. K. Rajamani. Relative completeness of abstraction refinement for software model checking. In *Proceedings of TACAS: Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *LNCS*. Springer Verlag, 2002.

2. Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Termination of polynomial programs. *Concurrency Theory (CONCUR)*, August 2005. To appear.
3. D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 2(5):323–342, April 1983.
4. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.
5. Michael Colón, Sriram Sankaranarayanan, and Henny Sipma. Linear invariant generation using non-linear constraint solving. In *CAV*, pages 420–432, 2003.
6. P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.
7. T. C. Nicholas Graham, Tore Urnes, and Roy Nejabi. Efficient distributed implementation of semi-replicated synchronous groupware. In *UIST '96: Proceedings of the 9th annual ACM symposium on User interface software and technology*, pages 1–10, New York, NY, USA, 1996. ACM Press.
8. G.J. Holzmann. *The Spin Model Checker - Primer and Reference Manual*. Addison-Wesley, 2004.
9. M. Kamel and S. Leue. Formalization and validation of the General Inter-ORB Protocol (GIOP) using Promela and Spin. *Software Tools for Technology Transfer*, 2:394–409, 2000.
10. S. Leue, R. Mayr, and W. Wei. A scalable incomplete boundedness test for UML RT models. In *Proc. of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS 2004*, Lecture Notes in Computer Science. Springer Verlag, 2004.
11. S. Leue, R. Mayr, and W. Wei. A scalable incomplete test for buffer overflow of Promela models. In *Proc. of the International SPIN Workshop on Model Checking of Software SPIN 2004*, volume 2989 of *Lecture Notes in Computer Science*. Springer Verlag, 2004.
12. Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, pages 239–251, 2004.
13. Andreas Podelski and Andrey Rybalchenko. Transition invariants. In *Proc. of LICS'2004: Logic in Computer Science*, pages 32–41. IEEE, 2004.
14. S. F. Siegel and G. S. Avrunin. Improving the precision of inca by eliminating solutions with spurious cycles. *IEEE Trans. Softw. Eng.*, 28(2):115–128, 2002.
15. M. H. ter Beek, M. Massink, D. Latella, and S. Gnesi. Model checking groupware protocols. In *COOP*, pages 179–194, 2004.
16. F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

Symbolic Model Checking for Asynchronous Boolean Programs

Byron Cook[1], Daniel Kroening[2], and Natasha Sharygina[3]

[1] Microsoft Research
[2] ETH Zurich
[3] Carnegie Mellon University

Abstract. Software model checking problems generally contain two different types of non-determinism: 1) non-deterministically chosen values; 2) the choice of interleaving among threads. Most modern software model checkers can handle only one source of non-determinism efficiently, but not both. This paper describes a SAT-based model checker for asynchronous Boolean programs that handles both sources effectively. We address the first type of non-determinism with a form of symbolic execution and fix-point detection. We address the second source of non-determinism using a symbolic and dynamic partial-order reduction, which is implemented inside the SAT-solver's case-splitting algorithm. The preliminary experimental results show that the new algorithm outperforms the existing software model checkers on large benchmarks.

1 Introduction

Model checking [1] is a formal verification technique for detecting behavioral anomalies in system descriptions. In recent years, a number of model checkers have been built specifically for the analysis of software. These tools have uncovered defects that would have otherwise gone undetected. However, they do not scale gracefully when applied to software of substantial size. Thus, much of the research on model checking has focused on improving scalability.

The size of the state space of a system is directly related to the amount of non-determinism present in the model. Concurrent software with asynchronous interleaving semantics has two sources of non-determinism: 1) Non-deterministic choice of data values, given explicitly in the program, and 2) the non-deterministic choice of the interleavings among the threads.

Powerful techniques have been developed to address both of these forms of non-determinism. Partial-order reduction is specifically designed to mitigate the concurrency among threads. Symbolic data structures concisely represent large sets of states. Unfortunately, these two techniques are difficult to combine. For this reason, with few exceptions, model checkers for software systems tend to come in one of two flavors: *Symbolic software model checkers* are strong when proving properties of programs with symbolic data but are not good at reasoning about concurrent programs with many threads; *Explicit-state model checkers* have powerful methods for the verification of programs with multiple threads,

P. Godefroid (Ed.): SPIN 2005, LNCS 3639, pp. 75–90, 2005.

but are not useful when applied to systems with significant amounts of symbolic data.

In this paper, we propose a model checking algorithm that efficiently analyzes programs with both non-deterministic data values and multiple threads of execution. The algorithm is limited to Boolean programs [2,3] extended with asynchronous threads [4,5]. Boolean programs—which are like C programs, but are limited to variables with type `bool`—have become a common model for tools that implement counterexample-guided abstraction refinement for software verification. Boolean programs allow the programmer to choose data values non-deterministically. We restrict ourselves to non-recursive programs, which we have found to be acceptable when performing analysis on system-level code. We also restrict the set of properties that can be verified to those that can be expressed in terms of reachability.

The algorithm described in this paper can be used immediately from within software model checkers such as SLAM [6] or BLAST [7]. These model checkers implement software predicate abstraction, i.e., they abstract a C program into a Boolean program. Using SLAM, we can now verify properties of device drivers with an accurate representation of the threads together with abstract representations of their environments.

The contribution of this paper is a method for combining SAT-based symbolic model checking and the partial-order reduction. We represent the states symbolically using a parametric representation [8,9]. The data structure grows linearly in the number of execution steps, even in the presence of non-deterministically chosen data values. As the parametric representation is not canonical, the fixpoint detection becomes harder. We use solvers for Quantified Boolean Formulae (QBF) for this task. We leverage the recent remarkable improvements in this technology [10,11].

We use a propositional logic SAT-solver as part of the symbolic simulation algorithm. This allows us to implement a form of partial-order reduction as a modification of the SAT-solver. The key idea behind this method is that the case-splitting algorithm used within backtracking-based SAT-solvers can be modified to eliminate undesired interleavings. This turns out to be much faster than alternative combination methods, such as adding constraints to the query that is passed to the SAT-solver. The resulting reduction is dynamic, as the choice of interleaving depends on the particular set of states found during the reachability analysis. We have implemented the algorithm proposed in this paper in a tool called BOPPO.

The remainder of this paper is organized as follows. We provide some background on Boolean programs in Section 2. We then describe our algorithm in the Sections 3 and 4. We describe the results for our experimental evaluations in Section 5. In Section 6, we conclude and discuss some ideas for future work.

Related Work. Several model checkers support sequential Boolean programs. BEBOP [2] and MOPED [3] are BDD-based symbolic model checkers, and both handle recursive procedures. In principle, because BOPPO supports only a fixed number of threads and non-recursive procedures, the threaded programs could

be converted into sequential programs that BEBOP and MOPED could process. This is not practical, however, because only a lightweight and static form of partial-order reduction could be applied during the translation, rather then the dynamic one that BOPPO employs.

DIZZY [12] uses SAT-based symbolic simulation. The fix-point detection is done by computing BDDs representing the set of reachable states. Our work uses a similar algorithm, but uses QBF for the fix-point detection. As BEBOP and MOPED, DIZZY does not support multiple threads.

Several previous efforts have also applied model checking to Boolean programs with asynchronous threads. For example, Jain, Clarke and Kroening [5] use the BDD-based model checker NUSMV [13] to verify concurrent Boolean programs with only very limited success.

Forms of partial-order reduction for explicit-state model checking (examples include [14,15]) have been a particularly effective for verifying programs and protocols with many threads. For example, Ball, Chaki and Rajamani [4] describe a partial-order reduction based explicit state model checker, called BEACON, for asynchronous Boolean programs. BEACON, however, was overly sensitive to the occurrence of symbolic data generated by SLAM.

The idea of combining symbolic reasoning with partial-order reduction is not new. Our proposal shares a great deal of motivation with Alur *et al.* [16], who describe a method of combining partial-order reduction together with a BDD-based symbolic model checker. Their algorithm first computes a constrained transition relation, called an *ample transition relation*. This is then given to a BDD-based model checker. Our experiments indicate that this technique does not provide much benefit in the context of SAT-solvers. The overhead of adding static constraints to the SAT-solver's data structure seems to abate the potential benefit of less state-space exploration. As it turns out, many of the constraints that are added are actually never used, resulting in wasted effort. Our implementation, which simply limits the assignments from which the SAT-solver can choose when case-splitting, requires less overhead when computing representative paths. In [17], the reduction is applied before passing the model to a Bounded Model Checker (BMC). In [18], interleavings are added incrementally to a BMC instance. In contrast to our work, a fix-point is not detected, and thus, the algorithm is incomplete.

In [19], Lerda, Sinha and Theobald integrate partial-order reduction into a BDD-based model checker, as opposed to a pre-processing step. This approach is similar to our proposal. The difference between this previous work and our proposal is in the representations of data, the class of solvers used, and methods of implementing the dynamic partial-order reduction. Whereas they use BDDs, we use SAT and QBF solvers and must therefore implement the partial-order reduction within the SAT-solver in a different manner.

Several methods address the problem of scalability in the presence of threads and non-deterministically chosen data via forms of decomposition [20,21]. These techniques usually either sacrifice some amount of completeness or require small amounts of intervention from the user. The advantage of these approaches is that

the analysis is much more scalable. In the future, researchers interested in thread modular approaches may be able to use our method of combining partial-order reduction and symbolic reachability in a way that allows them to improve on the completeness and user-interaction required.

Unsound approaches have also proved successful in finding bugs in concurrent programs. For example, Qadeer & Rehof [22] note that many bugs can be found when the analysis is limited to execution traces with only a small set of context-switches. This analysis supports recursive programs. Our approach complements these techniques because, while they are unsound, they are able to analyze a larger set of programs.

2 Boolean Programs

2.1 Boolean Programs and Predicate Abstraction

Predicate abstraction [23,24] is a commonly used method for systematically constructing conservative abstractions of software. When combined with reachability analysis and an automatic abstraction refinement mechanism, it forms an effective model checking strategy. Predicate abstraction constructs the abstraction by tracking only certain predicates on the data. Each predicate is represented by a Boolean variable in the abstract program, while the original data variables are eliminated. Extra non-determinism is added into the abstraction in order to maintain soundness of the sequential control-flow constructs in the abstraction. When predicate abstraction is performed on software systems with threads, the result is an abstraction that makes fundamental use of both non-deterministically chosen values and non-deterministically scheduled threads. Therefore, we need an efficient reachability analysis for these abstract models.

The following example shows code that is typical of a Windows device driver:

```
void DecrementIo(DEVICE_OBJECT *  DeviceObject) {
    EXT * ext = (EXT*)DeviceObject->DeviceExtension;
    int IoIsPending = InterlockedDecrement (&ext->IoIsPending);
    if (!IoIsPending){ KeSetEvent (&ext->event, IO_NO_INCREMENT, FALSE); }
}
```

An abstraction of this function is obtained by passing it to SLAM [6]. In the first iteration of the abstraction refinement loop, SLAM computes the following Boolean program fragment:

```
void DecrementIo_abstraction() {
    InterlockedDecrement_abstraction();
    goto L1,L2;
    L1: KeSetEvent_abstraction();
    L2: return;
}
```

This example demonstrates how predicate abstraction generates Boolean programs that make non-trivial use of both forms of non-determinism. This abstrac-

tion is using a non-deterministic `goto` instruction to model the conditional operator in the original function. This code fragment is also calling an abstraction of the Windows kernel synchronization primitive `KeSetEvent`.

In further refinement iterations, SLAM usually adds variables to the abstraction. Suppose the following predicates are used to refine the abstraction above:

$$\{\ \texttt{b1} \triangleq \texttt{ext} == \&\texttt{envext},\ \texttt{b2} \triangleq \texttt{envext.IoIsPending} == 1$$
$$,\ \texttt{b3} \triangleq \texttt{envext.IoIsPending} == 2,\ \texttt{b4} \triangleq \texttt{IoIsPending} == 2$$
$$,\ \texttt{b5} \triangleq \texttt{IoIsPending} == 1,\ \texttt{b6} \triangleq (*\texttt{ext}).\texttt{IoIsPending} == 1$$
$$,\ \texttt{b7} \triangleq (*\texttt{ext}).\texttt{IoIsPending} == 2\}$$

This results in the following new abstract model:

```
bool b1,b2,b3;
void DecrementIo_abstraction() {
    bool b4,b5,b6,b7;
    b1,b6,b7 = *,*,*
     constrain((!(b1' && b2) || b6') && (!(b1' && b3) || b7'));
    b4,b5 = InterlockedDecrement_abstraction(b6,b7);
    goto L1,L2;
    L1: assume(!b4 && !b5);
        KeSetEvent_abstraction();
    L2: return;
}
```

Due to the imprecision of the abstraction, we cannot prove that `ext==&envext`, nor can we prove that `ext!=&envext`. Therefore, a non-deterministically chosen value has to be assigned to the variable `b1`, which represents this predicate. This is necessary to preserve the soundness of the analysis.

Furthermore, using the `constrain` operator, this assignment statement restricts the choice such that `b6` must be true after the assignment if `b1` is true after the assignment and `b2` is true before the assignment. Analogously, `b7` must be true after the assignment if `b1` is true after the assignment and `b7` is true before the assignment. This abstraction also refines the non-deterministic `goto` using an `assume` statement: the program declares that any transition passing through the `L1` location must ensure that `b4` and `b5` are false.

2.2 Formal Semantics of Boolean Programs

In this section, we provide a simple operational semantics for asynchronous, concurrent Boolean programs. Later, in Section 3.2, we use the semantics to construct an algorithm that transforms Boolean program reachability into a propositional logic formula. The formalization is based on the description of sequential Boolean programs in [2].

Definition 1. *An* explicit state η *of a Boolean program is a tuple* (i, Ω)*, with* $i : T \mapsto L$ *and* $\Omega : V \mapsto \mathbb{B}$*.*

The first component of an explicit state η, called i, is a mapping from the set of threads T into the set of program locations L. Thus, $i(t)$ denotes the

instruction that is to be executed next by thread $t \in T$. The second component, called Ω, is a mapping from the set of variables V into the set of the two Boolean values, i.e., it assigns an explicit value to each state variable.

Notation. Given a valuation Ω and an expression e over the variables V, we use $\Omega(e)$ in order to denote the evaluation of e. This is defined in the usual way. In addition to that, we also allow expressions that refer to the values of variables in two different states η_1 and η_2. Syntactically, the values of the two states are distinguished by using primed versions of the variables. We use $(\eta_1, \eta_2)(e)$ in order to denote the evaluation of e in the states η_1 and η_2. The unprimed variables in e are substituted by the values given in η_1, while the primed variables in e are substituted by the values given in η_2. As an example, consider the valuation $\Omega_1 = \{(x, 1), (y, 0)\}$ and $\Omega_2 = \{(x, 0), (y, 0)\}$. For these valuations, and an expression $e = x \vee x'$, we have $(\eta_1, \eta_2)(e) = 1 \vee 0$.

We also allow additional choice variables $\iota_1, \ldots, \iota_k$ inside the expressions. We use ι to denote the vector of these variables. Given a particular non-deterministic choice ι and a state η, we denote the evaluation of the expression e in η with the choice ι as $(\eta, \iota)(e)$.

Given an explicit state η, we denote the first component by $\eta.i$, and the second component by $\eta.\Omega$. For any function $f : D \to T$, we define $f[d/r] : D \to R$ as $f[d/r](x) = r$ if $d = x$, and $f[d/r](x) = f(d)$ otherwise.

Execution Semantics. Assume the scheduler picks thread $t \in T$ to execute in state η. We use $\eta_1 \to_t \eta_2$ to denote the fact that a transition from state η_1 is made to η_2 by executing one statement of thread t. The statement that is executed is $P(\eta_1.i(t))$. The relation $\eta_1 \to_t \eta_2$ is defined by a case-split on this instruction. The conditions for each statement are shown in Table 1. We explain the formalization of each statement as follows:

- The `skip` statement increments the program counter of thread t. The values of the variables and the program counters of the other threads do not change.
- The `goto` $\theta_1, \ldots, \theta_k$ statement changes the program counter of thread t to one of the program locations $\theta_1, \ldots, \theta_k$ given as argument. The choice is arbitrary, i.e., non-deterministic. The values of the variables and the program counters of the other threads do not change.
- The `assume` e statement behaves like `skip`, but with the additional constraint that the expression e must evaluate to `true` in state η_1. If the expression evaluates to false, η_1 has no successor states.
- The constrained assignment statement $x_1, \ldots, x_k := e_1, \ldots, e_k$ `constrain` e changes the program counter like `skip`. It also updates the values of the variables using the expressions $e_1, \ldots, e_k$. The expressions are evaluated in state η_1. The expressions may contain choice variables $\iota_1, \ldots, \iota_k$. These variables allow a non-deterministic choice on data, and are quantified existentially.
  The transition also has an additional constraint e. The constraint e is a predicate in terms of the current state η_1 and the next state η_2. It is evaluated in both states accordingly, where the next state variables are primed. If there is no choice for ι, which satisfies the constraint, state η_1 has no successor states.

Table 1. Conditions on the explicit state transition $\langle i_1, \Omega_1\rangle \rightarrow_t \langle i_2, \Omega_2\rangle$, for each type of statement $P(i_1)$

$P(i_1)$	i_2	Ω_2
`skip`	$i_2(x) = i_1[t/i_1(t)+1]$	$\Omega_2 = \Omega_1$
`goto` $\theta_1, \ldots, \theta_k$	$i_2(x) = i_1[t/\theta_1] \vee \ldots \vee$ $i_2(x) = i_1[t/\theta_k]$	$\Omega_2 = \Omega_1$
`assume` e	$i_2(x) = i_1[t/i_1(t)+1]$	$\Omega_2 = \Omega_1 \wedge$ $\Omega_1(e) = \mathsf{true}$
$x_1, \ldots, x_k := e_1, \ldots, e_k$ `constrain` e	$i_2(x) = i_1[t/i_1(t)+1]$	$\exists \iota.\ \Omega_2 = (\Omega_1[x_1/(\Omega_1, \iota)(e_1)]$ $\ldots [x_k/(\Omega_1, \iota)(e_k)] \wedge$ $(\eta_1, \eta_2, \iota)(e)$

We do not define semantics for syntactic sugar such as `if` or `while`, as these statements can be easily transformed using `goto` and `assume`, as illustrated in section 2.1. Also, function calls can be inlined; we do not support unbounded recursion, as the reachability problem for concurrent programs with unbounded recursion is undecidable.

Finally, we write $\eta_1 \rightarrow \eta_2$ if there exists a thread $t \in T$ such that $\eta_1 \rightarrow_t \eta_2$. We say that there is a transition from η_1 to η_2 in this case, or that η_1 is reachable from η_2 with one transition.

A state η_2 is reachable from a state η_1 in k transitions if there exists a state η', η' is reachable from η_1 in $k-1$ transitions, and η_2 is reachable from η' in one transition. Given an initial state η_I, the set of reachable states is the set of states that is reachable from η_I in any number of transitions. The property we check is reachability of states with particular program locations.

3 SAT-Based Symbolic Simulation

In this section we describe how we represent a set of states symbolically using formulae, and then how to transform Boolean programs into such formulae.

3.1 Representation of States

Definition 2. *A symbolic formula is defined using the following syntax rules:*

1. *The Boolean constants* true *and* false *are formulae.*
2. *The non-deterministic choice variables* $\iota_1, \ldots$ *are formulae.*
3. *If* f_1 *and* f_2 *are formulae, then* $f_1 \wedge f_2$, $f_1 \vee f_2$, *and* $\neg f_1$ *are formulae.*

The set of such formulae is denoted by $\mathcal{F}$.

A symbolic formula may evaluate to multiple values due to the choice variables. As an example, the pair of formulae $\langle \iota_1, \iota_2 \wedge \neg \iota_1\rangle$ may evaluate to $\langle 0, 0\rangle$, $\langle 1, 0\rangle$, $\langle 0, 1\rangle$, but not to $\langle 1, 1\rangle$. We use these symbolic formulae in order to represent a set of states:

Definition 3. *A* symbolic state σ *is a triple* $\langle i, \omega, \gamma \rangle$*, with* $i : T \mapsto L$*,* $\omega : V \mapsto \mathcal{F}$*, and* $\gamma : \mathcal{F}$*.*

Given a particular valuation for the choice variables ι, we denote the value of a symbolic formula f as $\iota(f)$.

The first component of a symbolic state σ, called i, is identical to the first component of an explicit state (definition 1). The second component, called ω, is a mapping from the set of variables V into the set of formulae. It denotes the *symbolic* valuation of the state variables. The third component, called γ, is a formula that represents the guard of the state symbolically.

Thus, we represent the program counters explicitly, while the program variables are represented symbolically. The set of explicit states represented by σ are those states η that satisfy the following conditions:

- They have the same PC values given by i.

$$\eta.i = \sigma.i \tag{1}$$

- There exists a non-deterministic choice ι, which satisfies the guard γ, and assigns values to the variables that match the values given by Ω.

$$\exists \iota.\iota(\gamma) \ \wedge \ \forall v \in V.\Omega(v) = \iota(\omega(v)) \tag{2}$$

Thus, the set of explicit states corresponding to a symbolic state is defined using a predicate in the parameter ι. Thus, we have a *parametric representation.* Parametric representations of sets of states have been used in formal verification before [8,9], but mostly in the context of hardware verification.

Note that the problem of whether there exists an explicit state represented by a given symbolic state is equivalent to the problem of propositional satisfiability. A satisfying assignment contains concrete valuations for the state variables and for the choice variables, and thus, a SAT-solver provides a witness.

3.2 Symbolic Execution

Assume that the scheduler picks thread $t \in T$ to execute in the symbolic state σ. In analogy to the explicit state model, we use $\sigma_1 \rightarrow_t \sigma_2$ to denote the fact that a transition from state σ_1 is made to σ_2 by executing one statement of thread t. Again, the statement that is executed is $P(\sigma_1.i(t))$. The definition of the relation $\sigma_1 \rightarrow_t \sigma_2$ is done using a case-split on this instruction. The conditions for each statement are shown in Table 2. The column describing the constraints on the program counters i_1 and i_2 is identical to the column in Table 1, and therefore is not repeated here. We explain the formalization of each statement as follows:

- The definitions of the `skip` and `goto` statements follow the definitions for the explicit state case. The formulae for the guards are not changed by these statements.
- In the symbolic case, the `assume` e statement does not have the precondition that e is true. Instead, the condition e is instantiated in the state σ_1. This results in a symbolic formula. The symbolic formula is conjoined with the guard γ_1, forming the formula γ_2.

Table 2. Conditions on the symbolic transition $\langle i_1, \omega_1, \gamma_1 \rangle \rightarrow_t \langle i_2, \omega_2, \gamma_2 \rangle$, for each type of statement $P(i_1)$. For the constraints on i_1 and i_2, see table 1.

$P(i_1)$	ω_2	γ_2
`skip`	$\omega_2 = \omega_1$	$\gamma_2 = \gamma_1$
`goto` $\theta_1, \ldots, \theta_k$	$\omega_2 = \omega_1$	$\gamma_2 = \gamma_1$
`assume` e	$\omega_2 = \omega_1$	$\gamma 2 = (\gamma_1 \wedge \omega_1(e))$
$x_1, \ldots, x_k := e_1, \ldots, e_k$ `constrain` e	$\omega_2 = (\omega_1[x_1/\omega_1(e_1)] \ldots [x_k/\omega_1(e_k)]$	$\gamma 2 = (\gamma_1 \wedge (\omega_1, \omega_2)(e))$

– In the symbolic case, a constrained assignment statement $x_1, \ldots, x_k := e_1, \ldots, e_k$ `constrain` e updates the values of the variables using the expressions $e_1, \ldots, e_k$. The expressions are evaluated in state η_1. It is no longer necessary to instantiate the values of the non-deterministic choice variables ι, as $\omega(v)$ is now a formula, and not a Boolean value. Thus, the choice variables become part of the formula. Also, the additional constraint e is added to the guard, in analogy to an `assume` statement.

3.3 Reachability Algorithm

In order to check reachability of a particular program location $b \in L$ using the symbolic model, we implement an exhaustive search of the state space. This is done by most explicit state model checkers as well, e.g., by SPIN [25]. The basic algorithm is shown in Figure 1. The main differences between our implementation and an explicit state model checker are as follows:

1) We maintain a queue of symbolic states for the search. A search heuristic picks the next state to explore from the queue.

2) Before reachability of a bad state σ can be concluded, we must run a SAT solver (denoted by the function ISSATISFIABLE) in order to check that $\sigma.\gamma$ is satisfiable, and thus, the set of concrete states represented by σ is non-empty. Note that the guards of the states on one path only get stronger, and never weaker, and thus, it is sufficient to check the guards of the bad states only.

3) In order to conclude that no bad states are reachable, explicit state model checkers maintain a history of the states that have been explored. This set of states is typically organized using a hash table. Because of the symbolic representation, we cannot use this approach. Instead, we use a symbolic solver in order to compare the symbolic state that is chosen next to explore with the states that have been explored so far. This is implemented in the procedure ISHISTORY. The details of this function are described in section 4.

3.4 Partial-Order Reduction

When computing the successors of a given symbolic state σ, we usually have to consider the possibility that any of the threads $t \in T$ can make a transition. The choice is non-deterministic. Formally, we have to compute all states σ' for which

```
    // Input: Boolean Program P with locations L, bad location b ∈ L
    // Output: true iff b is reachable in P
    // Variables: Queue Q of symbolic states
    SymbolicReachability(P, b)
 1  Compute initial state σ_I
 2  Q := {σ_I};
 3  while (¬Q ⊭ ∅)
 4        σ := Element from Q;
 5        if IsHistory(σ) then
 6              Q := Q \ σ;
 7        elseif ∃t ∈ T. σ.i(t) = b ∧ IsSatisfiable(σ.γ) then
 8              return true;
 9        else
10              Q := (Q \ σ) ∪ GetSuccessors(P, σ);
11        endif
12  end
13  return false;
```

Fig. 1. High Level Description of the Symbolic Reachability Algorithm

a thread $t \in T$ exists which can make a transition from σ to $state'$. A sequence of choices for a particular thread t is called an *interleaving*. The problem is that the number of states explored can grow dramatically with the number of threads. Even with just two threads, the number of interleavings blows up in the number of execution steps. In contrast to that, a sequential program only requires as many symbolic states as there are execution steps.

The purpose of *Partial-Order Reduction* [15] is to reduce the number of paths that have to be explored. This is done in a way that preserves the property, i.e., the property holds on the reduced model if and only if it holds on the full, original model.

Symbolic Partial-Order Reduction Using SAT. The approach we take is related to what many explicit state model checkers implement. We aim at finding a thread t that makes an *invisible* transition, i.e., a transition which is independent from a transition made by any other thread $t' \not= t$. We compute the sets of variables written and read by each of the threads. Let R_t denote the set of variables that are read, and W_t the set of variables that are written by thread t in the current state. If thread t is not enabled, these sets are empty. If a thread t is found with $W_t \cap (\bigcup_{i \neq t} R_i \cup W_i) = \emptyset$ and $R_t \cap \bigcup_{i \neq t} W_i = \emptyset$, we only explore the successors generated by executing t. All other transitions are discarded.

This reduction preserves the property we are checking, i.e., reachability of program locations. The computation of the reduction requires knowledge of the enabled transitions and of the dependencies between the transitions. This is computationally inexpensive in case of an explicit state model checker, as all the

values of the variables are known. In contrast, we use a symbolic representation. The question of whether a particular transition is enabled or not corresponds to a SAT instance. A syntactic over-approximation of the set of enabled transitions and the dependencies is feasible, but often does not result in a significant reduction. We therefore use a modified SAT solver in order to compute the set of interleavings we explore.

SAT has been used in the context of asynchronous transition systems before. As in most existing approaches, we build a SAT instance that has non-deterministically chosen variables for the thread selector and an encoding of the transitions out of the given state. Typically, constraints on the thread selector variables are added upfront in order to limit the possible choice of interleavings. However, our initial experiments showed that most of these constraints are unnecessary, as they eliminate transitions out of states that are unreachable, and often make the instance much harder.

We therefore use the following, alternative approach: the SAT instance we form uses a one-hot encoding for a thread about to make an invisible transition. We implement the constraints on the variables that are read and written as part of the case-splitting heuristic of ZChaff, and not by adding appropriate clauses, as this information is known statically. The SAT-solver only needs to determine which threads are enabled, i.e., have a satisfiable guard.

Once a local interleaving is found, it is explored. If no local interleaving is found, the thread to be executed is chosen by the SAT-solver's decision heuristic. Once its successors are computed, we add a blocking clause to prevent the same transition from being explored again and backtrack.

Cycle Detection. The method of removing interleavings that we described above could lead to unsound results. In fact, there is a possibility that some transitions will be delayed forever because of a cycle in the reduced model.

To prevent the loss of transitions, partial-order reduction techniques require satisfaction of a *cycle condition* [26]. The cycle condition prohibits cycles that contain a state in which some transition is enabled, but is never taken for any state on the cycle. The intuitive reason for this condition is to avoid postponing a transition indefinitely while generating the reduced model.

Algorithmically, we solve this issue in the same way as most explicit state model checkers: when postponing a transition, we note this fact on the search stack. If the IsHistory procedure detects that a state has been explored before, we resume the evaluation of the postponed transitions.

4 Fix-Point Detection

In order to detect fix-points, we need to compare the new set of states to the set of states that we have already explored. When using BDDs, two sets of states can be compared by simply comparing the graphs of the BDDs. The drawback of using BDDs is that already only very few steps of symbolic simulation may result in prohibitively large BDDs.

As described in the previous section, we store the states using a non-canonical symbolic representation. While this representation allows us to execute a statement symbolically in linear time, we pay a price in form of a harder fix-point detection problem.

The fix-point detection is implemented in the IsHistory procedure. It takes a new symbolic state σ_n as input and returns true if it is subsumed by an old symbolic state σ_o in a set H. The program counter part of the state is stored explicitly. Thus, the first step of the algorithm is to obtain the set of old states $H' \subseteq H$ with program counter values that match those of state σ_n. This is implemented using a hash table, as is done in most explicit state model checkers. The number of entries in this table is limited by the partial-order reduction. We therefore do not expect a blowup in this data structure.

The set H' corresponds to a *disjunctive partitioning* of the set of states. Disjunctive partitionings are commonly used in symbolic model checkers for asynchronous concurrent programs, e.g., in [13,27].

The second step is to check whether a symbolic state in $\sigma_o \in H'$ subsumes the symbolic state σ_n, i.e., if all explicit states represented by σ_n are also contained in σ_o. Note that we will not detect the case that σ_n is not covered by any single $\sigma_o \in H'$, but rather by a combination of states in H'. Comparing the new state with the union of the symbolic states in H' would be too expensive. This may delay the detection of the fix-point, but will neither affect soundness nor termination.

A state σ_n is subsumed by a state σ_o if for all explicit states represented by σ_n there exists an identical state represented by σ_o. As the program counter components already match, we only need to compare the values of the state variables. As given by Equation 2, the set of explicit states represented by a symbolic state is defined using an existential quantification over the choice variables ι. Formally, for each choice of inputs ι_n for the new state σ_n, there must exist a (possibly different) choice of inputs ι_o for the old state σ_o that results in the same state:

$$\forall \iota_n | \iota_n(\gamma_n). \quad \exists \iota_o | \iota_o(\gamma_o). \quad \iota_n(\omega_n) = \iota_o(\omega_o) \tag{3}$$

Equation 3 can be transformed into a Quantified Boolean Formula (QBF) and passed to a QBF solver such as Quantor [10] or Quaffle [11]. We have found that modern QBF solvers, and especially Quantor, can handle surprisingly large instances that we generate. If the QBF solver determines the formula to be true, we can discard the state σ_n. Otherwise, we insert σ_n into H, and proceed with the state space exploration using the successors of state σ_n.

Optimization. In Equation 3, the outermost quantification is done over the non-deterministic choice variables used as parameter for the states represented by σ_n. Given a deep symbolic simulation, the number of such variables may be large.

Note that we only care about the values of the state variables in a state represented by σ_n. Thus, we can re-write Equation 3 such that the outer quantification is done over the state bits, and not over the non-deterministic choices.

Table 3. Experimental results: n/a denotes that the model checker does not handle the benchmark due to lacking features, * denotes that the time limit (1 hour) or memory limit (2 GB) was exceeded

Benchmark	MOPED	SPIN	BEBOP	ZING	BOPPO
1	**0.1s**	*	0.1s	n/a	0.6s
2	*	**3.8s**	120s	n/a	27.0s
3	n/a	n/a	**0.17s**	n/a	0.43s
4	n/a	*	2058s	n/a	**75.6s**
5	n/a	*	n/a	*	**55.8s**

$$\forall x_n. \quad \exists \iota_n, \iota_o. x_n = \iota_n(\omega_n) \wedge (\iota_n(\gamma_n) \Longrightarrow (\iota_o(\gamma_o) \wedge \iota_n(\omega_n) = \iota_o(\omega_o)) \qquad (4)$$

The number of state-bits may be much smaller than the number of non-deterministic choices, and thus, the complexity of the formula is reduced.

Another simple optimization is to restrict the set of variables we consider to $V' \subseteq V$, where V' is the set of variables that are *active* in any of the program locations $L' \subseteq L$ given by any of the program counters.

A variable is active in a program location if its value is of relevance to any instruction reachable from the location. E.g., local variables that are not yet in scope can be disregarded when comparing the values of the state variables.

A third optimization is to partition the set of variables into groups $C_1, \ldots, C_k$ that share choice variables. Indirect sharing, through other variables, has to be considered.

5 Experimental Results

We have implemented the algorithm described above in a tool called BOPPO. We use Limmat as the SAT solver, and Quantor as the QBF solver.

In this section, we compare BOPPO with other model checkers. We use the explicit state model checkers SPIN [25] and ZING [28]. We also compare our BOPPO with MOPED [3] and BEBOP [2], which are BDD-based symbolic model checkers. Neither BEBOP nor MOPED supports multiple threads, however. The experimental results are summarized in Table 3.

Benchmarks 1-4 are sequential; the first two benchmarks are artificial and contain about 30 Boolean variables. In the first benchmark, most states are reachable. The symbolic model checkers BEBOP, BOPPO, and MOPED handle this benchmark easily, while the explicit state model checkers run out of memory even with such a small number of state bits. The second benchmark encodes a multiplication over the Boolean variables. SPIN handles this benchmark easily, while MOPED exceeds the 2GB memory limit.

The benchmarks 3-5 are generated by SLAM. The SLAM model checker implements counterexample guided abstraction refinement for C programs. Benchmark 3 is a summary of 572 individual, small sequential benchmarks; the times given for the benchmark denote the average runtime. On the small benchmarks, BEBOP outperforms BOPPO. Benchmark 4 is a large sequential device driver.

An experimental version of SLAM provides support for the verification of concurrent programs[4] In this mode, ZING is used as a replacement for SLAM's sequential reachability engine, BEBOP. Benchmark 5 is generated from a 4500 LOC Windows device driver with three threads in this manner.

As ZING is an explicit state model checker, it is not well-adapted to handle the larger Boolean programs that are produced by predicate abstraction. As discussed in Section 2, SLAM generates abstractions that make frequent use of non-deterministic choice. When SLAM is used to verify the correctness of Windows device drivers, we must also provide abstract representations of the kernel, other device drivers, and user-level applications. This environment adds a large amount of additional non-determinism. For this reason, SLAM in combination with ZING can process only relatively small model checking examples.

With BOPPO, SLAM is now able to solve much larger problems. ZING is unable to solve benchmark 6 after more than an hour of execution. BOPPO is able to solve the benchmark within a minute. We attempted to run this same benchmark using SPIN and NUSMV without any positive result.

Surprisingly, BOPPO appears to make a contribution for sequential programs as well. As we try to apply SLAM to more difficult properties and larger programs, BEBOP is sometimes the performance bottleneck. This problem is exacerbated by experiments where we have used a theorem prover that is accurate with respect to pointer arithmetic, bit-vectors, structures and unions [29] — this causes many additional Boolean variables to be added to the abstraction and also causes the logic used in the transition relation of the Boolean program to become more complicated. This puts additional strain on BEBOP.

In the worst case, the predicates can begin to resemble the arithmetic from the original C program. BOPPO, because its symbolic representation is based on SAT and QBF and not BDDs, is better able to scale to larger and more complicated sequential Boolean programs.

6 Conclusion and Future Work

Symbolic model checking and partial-order reduction are hard to combine. For this reason model checkers for software systems typically treat non-trivial amounts of symbolic data, or non-trivial numbers of threads, but not both. We have presented a SAT-based model checking approach that can be used to efficiently reason about the safety of Boolean programs with both symbolic data and multiple threads. This allows model checkers which abstract software into Boolean programs to verify multi-threaded programs.

The algorithm presented in this paper implements partial-order reduction using SAT. The reduction is based on a change to the case-splitting algorithm used within the SAT-solver. This implementation strategy turns out to be better than an approach in which constraints on the interleavings are encoded as part of the input to the SAT-solver.

[4] Thanks to Georg Weissenbacher and Jakob Lichtenberg.

As future work, we want to experiment with other techniques for checking state subsumption for parametric representations. In [30], the authors use a SAT solver to compute a new parametric representation from a set of constraints. The new parametric representation is canonical for a given variable ordering, and thus allows an efficient fix-point detection. We would also like to try our techniques for checking liveness properties and for checking equivalence of two programs.

References

1. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
2. Ball, T., Rajamani, S.K.: Bebop: A symbolic model checker for Boolean programs. In: SPIN 00: SPIN Workshop. LNCS 1885, Springer-Verlag (2000) 113–130
3. Esparza, J., Schwoon, S.: A BDD-based model checker for recursive programs. In: CAV. LNCS 2102, Springer-Verlag (2001) 324–336
4. Ball, T., Chaki, S., Rajamani, S.K.: Parameterized verification of multithreaded software libraries. In: TACAS, Springer-Verlag (2001)
5. Jain, H., Clarke, E., Kroening, D.: Verification of SpecC and Verilog using predicate abstraction. In: Proceedings of MEMOCODE 2004, IEEE (2004) 7–16
6. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In: IFM. (2004)
7. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL 02: Symposium on Principles of Programming Languages, ACM Press (2002) 58–70
8. Coudert, O., Madre, J.: A unified framework for the formal verification of sequential circuits. In: ICCAD, IEEE (1990) 78–82
9. Aagaard, M.D., Jones, R.B., Seger, C.J.H.: Formal verification using parametric representations of boolean constraints. In: DAC, ACM Press (1999) 402–407
10. Biere, A.: Resolve and expand. In: Proc. SAT'04. LNCS, Springer (2004)
11. Zhang, L., Malik, S.: Conflict driven learning in a quantified boolean satisfiability solver. In: ICCAD. (2002)
12. Leino, K.R.M.: A SAT characterization of Boolean-program correctness. In: SPIN. (2003)
13. A. Cimatti et al.: NuSMV 2: An opensource tool for symbolic model checking. In: CAV. (2002) 359–364
14. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: POPL 05: Symposium on Principles of Programming Languages, ACM Press (2005)
15. Holzmann, G., Peled, D.: An improvement in formal verification. In: Proc. Formal Description Techniques, FORTE94, Chapman & Hall (1994) 197–211
16. Alur, R., Brayton, R.K., Henzinger, T.A., Qadeer, S., Rajamani, S.K.: Partial-order reduction in symbolic state-space exploration. FMSD **18** (2001) 97–116
17. Jussila, T., Niemelä, I.: Parallel program verification using BMC. In: ECAI 2002 Workshop on Model Checking and Artificial Intelligence. (2002) 59–66
18. Grumberg, O., Lerda, F., Strichman, O., Theobald, M.: Proof-guided underapproximation-widening for multi-process systems. In: POPL, ACM Press (2005) 122–131
19. Lerda, F., Sinha, N., Theobald, M.: Symbolic model checking of software. In: Software Model Checking (SoftMC). ENTCS (2003)
20. Alur, R., Henzinger, T., Mang, F., Qadeer, S., Rajamani, S., Tasiran, S.: MOCHA: Modularity in model checking. In: CAV. LNCS. Springer (1998) 521–525

21. Henzinger, T.A., Jhala, R., Majumdar, R., Qadeer, S.: Thread modular abstraction refinement. In: CAV, Springer (2003) 262–274
22. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: TACAS 05: Tools and Algorithms for Construction and Analysis of Systems, Springer-Verlag (2005)
23. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In Grumberg, O., ed.: CAV. Volume 1254 of LNCS., Springer (1997) 72–83
24. Colón, M., Uribe, T.: Generating finite-state abstractions of reactive systems using decision procedures. In: CAV. Volume 1427 of LNCS., Springer (1998) 293–304
25. Holzmann, G.: The model checker SPIN. IEEE Trans. on Software Engineering **23** (1997) 279–295
26. Peled, D.: All from one, one for all: on model checking using representatives. In: In Proc.of CAV. (1993)
27. Barner, S., Rabinovitz, I.: Effcient symbolic model checking of software using partial disjunctive partitioning. In: CHARME. (2003) 35–50
28. T. Andrews et al.: Zing: Exploiting program structure for model checking concurrent software. In: CONCUR 2004. (2004)
29. Cook, B., Kroening, D., Sharygina, N.: Cogent: Accurate theorem proving for program verification. In Etessami, K., Rajamani, S.K., eds.: Proceedings of CAV 2005. Volume 3576 of Lecture Notes in Computer Science., Springer Verlag (2005)
30. Chauhan, P., Clarke, E., Kroening, D.: A SAT-based algorithm for reparameterization in symbolic simulation. In: DAC 2004, ACM Press (2004) 524–529

Improving Spin's Partial-Order Reduction for Breadth-First Search

Dragan Bošnački[1] and Gerard J. Holzmann[2]

[1] Eindhoven University of Technology,
Den Dolech 2, P.O. Box 513,
5612 MB Eindhoven, The Netherlands
[2] NASA/JPL Laboratory for Reliable Software,
California Institute of Technology,
4800 Oak Grove Drive
Pasadena, CA 91006

Abstract. We describe an improvement of the partial-order reduction algorithm for breadth-first search which was introduced in Spin version 4.0. Our improvement is based on the algorithm by Alur et al. for symbolic state model checking for local safety properties [1]. The crux of the improvement is an optimization in the context of explicit state model checking of the condition that prevents action ignoring, also known as the cycle proviso. There is an interesting duality between the cycle provisos for the breadth-first search (BFS) and depth first search (DFS) exploration of the state space, which is reflected in the role of the BFS queue and the DFS stack, respectively. The improved version of the algorithm is supported in the current version of Spin and can be shown to perform significantly better than the initial version.

1 Introduction

Partial-Order Reduction (POR) [17,14,4,15,18,2] is one of Spin's [6] primary weapons against the state explosion problem. The standard POR algorithm in Spin [7,8,9] assumes a depth-first search (DFS) exploration of the state space. Starting with version 4.0.0, Spin also supports a breadth-first search (BFS) exploration mode, which is enabled when the model checker is compiled with optional compile-time directive `-DBFS`. It is therefore attractive to develop an efficient version of the POR algorithm that can be compatible with BFS.

In this paper we describe an improvement of the initial BFS version of the POR algorithm in Spin for the verification of safety properties, which achieves a reduction of the state space that is comparable to the DFS case, while still preserving the benefits of BFS exploration (e.g., finding the shortest counterexample to a correctness property). The improvement we describe is inspired by Alur et al.'s algorithm [1] for the application of POR in symbolic state space exploration.

The crucial novelty is a new version of the so called cycle proviso which prevents action ignoring. Unlike the full state space exploration, POR expands

P. Godefroid (Ed.): SPIN 2005, LNCS 3639, pp. 91–105, 2005.

only a subset of the enabled actions/transitions in a given state, called the ample set. The actions outside the ample set are temporarily ignored. However, if one is not careful, an action could be permanently ignored along some cycle in the reduced state space. Consider a state s that appears in both the full and the reduced state space. An action a is (permanently) ignored if it is executed in s in the full state space, but it is ignored along all execution sequences starting at s in the reduced state space.

To prevent this, the initial BFS POR algorithm in Spin required that each ample set must satisfy a special version of the cycle proviso: at least one state which is obtained as a result of an action from the ample set must appear outside the set of previously visited states.

Based on the theory in [1] we show that this condition can be weakened such that one does not forbid previously visited states that are still in the BFS queue (i.e., whose successors have not been explored yet). The intuition is that the ignoring problem is postponed until such states are revisited during the BFS. From them the execution of some already postponed state could be further postponed, but because of the finiteness of the state space all postponed transitions will eventually be executed. Unlike states in the BFS queue, visited states outside the queue will not be explored again. As a result, a cycle closed back through such states can lead to indefinite postponement. Experimental results confirm that the new proviso gives much better reductions than the old one, often comparable to and in some cases better than the reductions obtained with the standard DFS POR.

Related work. The POR algorithm of Alur et al. [1] is for symbolic state space exploration and as such it is based on BFS. However, as in the symbolic approach one works with sets of states instead of with individual states, a direct translation of the algorithm would result in a less efficient version than the one presented here. In particular, it is not clear what the analogue would be in an explicit (enumerative) state search of the set of most recently generated states ("front states") in [1].

For instance, in [12] a direct adaptation resulted in a proviso that forbids all previously visited states, similar to Spin's initial BFS cycle proviso discussed above. The implementation described in [12] is not directly comparable with the initial Spin implementation though, since the former works with a combination of depth and breadth-first searches which are applied interchangeably.

In another work [11], the authors exploit the fact that the concurrent systems we work with are defined by a parallel composition of sequential processes. This leads to the formulation of a static version of the cycle proviso, i.e., one which is enforced at compile time. The observation is that the existence of a cycle in the global state space implies the existence of a local cycle in one of the component processes. To break global cycles it suffices to break their local components. The algorithm now marks at least one transition in each local cycle as "sticky" to ensure that at least one state of a global cycle is fully expanded. This static condition is in general much stronger, and should therefore be expected to be less efficient, than our version of the proviso.

2 Preliminaries

This section introduces the concepts and terminology used in the paper. We also discuss the standard (DFS) version of the partial-order algorithm in Spin.

2.1 Transition Systems

To formally reason about state spaces, we introduce the notion of a *labeled transition system*.

Definition 1 (Labeled transition system). *A labeled transition system (LTS), is a 6-tuple* $(S, \hat{s}, A, \tau, \Pi, L)$*, where*

- S *is a finite set of* states*;*
- $\hat{s} \in S$ *is the* initial *state;*
- A *is a finite set of* actions*;*
- $\tau : S \times A \rightarrow S$ *is a (partial)* transition function*;*
- Π *is a finite set of boolean* propositions*;*
- $L : S \rightarrow 2^{\Pi}$ *is a* state labeling function*.*

Let $\mathcal{T} = (S, \hat{s}, A, \tau, \Pi, L)$ be an LTS. An action $a \in A$ is said to be $\mathcal{T}$-*enabled* in state $s \in S$, denoted $s \xrightarrow{a}_{\mathcal{T}}$ iff $\tau(s, a)$ is defined. The set of all actions $a \in A$ enabled in state $s \in S$ is denoted $enabled_{\mathcal{T}}(s)$; that is, for any $s \in S$, $enabled_{\mathcal{T}}(s) = \{a \in A \mid s \xrightarrow{a}_{\mathcal{T}}\}$. When the LTS is clear from the context we omit the $\mathcal{T}$ subscript. A state $s \in S$ is a *deadlock* state iff $enabled(s) = \emptyset$.

Transition function τ of LTS $\mathcal{T}$ induces a set $T \subseteq S \times A \times S$ of transitions defined as $T = \{(s, a, s') \mid s, s' \in S \wedge a \in A \wedge s' = \tau(s, a)\}$. To improve readability, we write $s \xrightarrow{a} s'$ for $(s, a, s') \in T$.

An *execution sequence* of LTS $\mathcal{T}$ is a (finite) sequence of consecutive transitions in T. For any natural number $n \in \mathbb{N}$, states $s_i \in S$ and actions $a_i \in A$ with $i \in \mathbb{N}$ and $0 \leq i < n$, $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots s_{n-1} \xrightarrow{a_{n-1}} s_n$ is called an execution sequence of length n of $\mathcal{T}$ iff $s_i \xrightarrow{a_i} s_{i+1}$ for all $i \in \mathbb{N}$ with $0 \leq i < n$. State s_n is said to be *reachable* from state s_0. A state is said to be reachable in $\mathcal{T}$ iff it is reachable from $\hat{s}$.

2.2 Partial-Order Reduction - Theoretical Framework

The basic idea of state space reduction is to restrict the part of the state space of a concurrent system that is explored during verification in such a way that all properties of interest are preserved. *Partial-order* reduction exploits the independence of properties from the many possible interleavings of the individual process actions of a concurrent system. In our context, actions correspond to Promela statements. Thus, partial-order reduction uses the fact that state-space explosion is often caused by the many possible interleavings of independent statements (actions) of concurrently executing processes. (For more details about the relation of the Promela models and their corresponding LTSs see, for instance, [8,6].)

To be practically useful, a reduction of the state space must be achieved on-the-fly, during the construction and traversal of the state space. This means that it must be decided *per state* which transitions, and hence which subsequent states, must be considered. Let $\mathcal{T} = (S, \hat{s}, A, \tau, \Pi, L)$ be some LTS.

Definition 2 (Reduction). *For any so-called* reduction *function* $r : S \to 2^A$, *we define the (partial-order)* reduction *of* $\mathcal{T}$ *with respect to* r *as the smallest LTS* $\mathcal{T}_r = (S_r, \hat{s}_r, A, \tau_r, \Pi, L_r)$ *satisfying the following conditions:*

- $S_r \subseteq S$, $\hat{s}_r = \hat{s}$, $\tau_r \subseteq \tau$, *and* $L_r = L \cap (S_r \times \Pi)$;
- *for every* $s \in S_r$ *and* $a \in r(s)$ *such that* $\tau(s, a)$ *is defined,* $\tau_r(s, a)$ *is defined.*

Note that these two requirements imply that, for every $s \in S_r$ *and* $a \in A$, *if* $\tau_r(s, a)$ *is defined, then also* $\tau(s, a)$ *is defined and* $\tau_r(s, a) = \tau(s, a)$.

Formally, if the function $r(s)$ is fixed in advance, the reduced LTS $\mathcal{T}_r$ is independent of the particular algorithm with which it is generated. In practice $r(s)$ is computed on-the-fly during the generation of $\mathcal{T}_r$, so the latter may depend on the algorithm. Viewing the LTS as a graph, we consider two cases: a depth-first and a breadth-first graph traversal algorithm.

It will be clear that not all reductions preserve all properties of interest. Depending on the properties that a reduction must preserve, we have to define additional restrictions on r. To this end, we need to formally capture the notion of independence. Actions occurring in different processes can easily influence each other, for example, when they access global variables.

The following notion of independence defines the absence of such mutual influence.

Definition 3 (Independence of actions). *Actions* $a, b \in A$ *with* $a \neq b$ *are* independent *in a given state* $s \in S$ *iff the following holds:*

- *if* $a \in enabled(s)$ *then* $b \in enabled(s)$ *iff* $b \in enabled(\tau(s, a))$,
- *if* $b \in enabled(s)$ *then* $a \in enabled(s)$ *iff* $a \in enabled(\tau(s, b))$, *and*
- $\tau(\tau(s, a), b) = \tau(\tau(s, b), a)$

Given a set $S' \subseteq S$, *we say that two actions* $a, b \in A$ *are* conditionally *independent (on* S'*) iff they are independent in all states* $s \in S'$. *If* $S' = S$, *we say that* a, b *are* unconditionally (globally) independent.

A typical example of independent actions are actions that correspond to assignments to or evaluations of local variables in distinct processes. Another case is two i/o operations on the same channel q under certain conditions: send and receive are independent provided that the channel q is neither empty nor full. Actions that are not (conditionally or unconditionally) independent are called (conditionally or unconditionally) dependent.

The first property we are interested in proving is absence of deadlock. In order to preserve deadlock states in a reduced LTS, the reduction function r must satisfy the following conditions:

- C0a: if $a \in r(s)$ then $a \in \mathit{enabled}(s)$
- C0b: $r(s) = \emptyset$ iff $\mathit{enabled}(s) = \emptyset$.
- C1 (persistence): For any $s \in S$ and execution sequence $s_0 \stackrel{a_0}{\rightarrow} s_1 \stackrel{a_1}{\rightarrow} \ldots \stackrel{a_{n-1}}{\rightarrow} s_n$ of length $n \in \mathbb{N} \setminus \{0\}$ such that $s_0 = s$ and $a_i \not\in r(s)$ for all $i \in \mathbb{N}$ with $0 \leq i < n$, it holds: action a_{n-1} is independent in s_{n-1} with all actions in $r(s)$.

The basic idea behind the persistence condition is that, during the state-space traversal, transitions caused by actions that are independent of all the actions chosen by the reduction function can be temporarily ignored, i.e., postponed. Action sets which satisfy conditions C0a, C0b, and C1 are called *persistent* sets [4]. Similarly, the corresponding function r is called a persistent function.

Theorem 1 (Deadlock preservation [4, Theorem 4.3]**).** *Let r be a reduction function for LTS $\mathcal{T}$ that satisfies conditions C0a, C0b and C1. Any deadlock state reachable in $\mathcal{T}$ is also reachable in the reduced LTS $\mathcal{T}_r$ and vice versa.*

The above mentioned Theorem 4.3 in [4] does not state that any deadlock reachable in a reduced LTS is also reachable in the original LTS. However, this result follows immediately from proviso C0b. Several authors have presented state-space-reduction algorithms that preserve deadlocks [13,16,5].

The second class of properties we discuss is the class of safety properties which includes Promela assertions [6]. The main obstacle in the verification of safety properties is the so called *action ignoring problem* which was identified for the first time in [17]. Informally, the ignoring problem occurs when a reduction of a state space ignores the actions of an entire process. For instance, if there is a cyclic process in the system which contains only globally independent actions, i.e., does not interact with the rest of the system, the reduction algorithm could ignore the rest of the system by choosing only actions of this process in $r(s)$. An action a is ignored in a state $s \in S_r$ iff $a \in \mathit{enabled}_{\mathcal{T}}(s)$ and for all s' which are reachable in $\mathcal{T}_r$ from s it holds $a \not\in \mathit{enabled}_{\mathcal{T}_r}(s')$. An action is ignored in $\mathcal{T}_r$ iff it is ignored in some state $s \in S_r$.

To avoid the ignoring problem we use a witness function W which enumerates the states in S_r such that we are sure that the ignoring of an action will stop at some point [1]. Let $\mathcal{T}$ be an LTS with a reduction function r. A mapping $W : S_r \rightarrow \mathbb{N}$ (from the set of states of the reduction $\mathcal{T}_r$ to the set of natural numbers) is a *witness* for r iff for all states $s \in S_r$ the following holds: if $r(s) \neq \mathit{enabled}(s)$, then there exists an action $a \in r(s)$ and a state $s' \in S_r$ such that $s \stackrel{a}{\rightarrow} s'$ and $W(s') < W(s)$. Thus, we introduce the following additional condition on $r(s)$:

- C2w (avoiding action ignoring using witness): The function r has a witness W.

By conditions C0a, C0b, and C1 an action a, which is enabled in $\mathcal{T}$ in some s which is also in $\mathcal{T}_r$, cannot be disabled by any action in $r(s)$. Hence, by C2w, there is always a descendant s' of s, such that $a \in \mathit{enabled}_{\mathcal{T}}(s')$ and $W(s') < W(s)$. Continuing this argument further on s' and its descendants we can obtain a

strictly decreasing sequence of naturals $W(s), W(s'), \ldots$. As we work with finite state spaces, we will eventually arrive in some state s'' in which we cannot satisfy $C2w$, because $W(s'') < W(s''')$, for any immediate descendant s''' of s''. Obviously in such a state $r(s'') = enabled(s'')$ contains all delayed actions which (because of the persistence) remain enabled along the way and thus they are not ignored in s''.

The fact that any enabled transition in a given state s of the reduced state space will be eventually executed in some state reachable from s implies that each execution sequence σ starting in s has a representative in the reduced state space. If we see the execution sequence as a sequence of actions, this representative is a permutation of an action sequence obtained by extending σ with another (possibly empty) action sequence σ' from the original state space. More formally, the claim is given by the following theorem:

Theorem 2. *Given an LTS $\mathcal{T}$ and a reduction function r that satisfies C0a, C0b, C1, and C2w, let $s_0 \stackrel{a_0}{\rightarrow} s_1 \stackrel{a_1}{\rightarrow} \ldots s_{n-1} \stackrel{a_{n-1}}{\rightarrow} s_n$ be a finite execution sequence of $\mathcal{T}$, such that $s_0 \in S_r$. Then there exists (in $\mathcal{T}$) an execution sequence $s_n \stackrel{a_n}{\rightarrow} s_1 \stackrel{a_{n+1}}{\rightarrow} \ldots s_{n+k-1} \stackrel{a_{n+k-1}}{\rightarrow} s_{n+k}$, $(k \geq 0)$, such that in $\mathcal{T}_r$ there exists an execution sequence $s_0 \stackrel{a_{\pi(0)}}{\rightarrow} s'_1 \stackrel{a_{\pi(1)}}{\rightarrow} \ldots s'_{n+k-1} \stackrel{a_{\pi(n+k-1)}}{\rightarrow} s_{n+k}$, where $a_{\pi(0)}, a_{\pi(1)}, \ldots, a_{\pi(n+k-1)}$ is a permutation of $a_0, a_1, \ldots, a_{n+k-1}$.*

Proof of the above theorem can be found in [1]. Analogous results were proven previously using different versions of the condition that prevents action ignoring (e.g. [17,4]).

Theorem 2 is a meeting point of almost all existing POR-like techniques. It implies preservation of various classes of safety properties (for instance, see [18] for an overview.) Among them are also Promela assertions that can be fitted in a straightforward way in one of the existing approaches like assertions in the sense of [4,7], fact transitions of [17], or local properties of [1].

2.3 The Standard Partial-Order-Reduction Algorithm of Spin

Given the theorems of the previous subsection, the challenge is to find interesting reduction functions and efficient algorithms implementing the corresponding reductions. The standard partial-order-reduction algorithm of Spin is described in [8,14]. The most important aspects of the algorithm are the following: (1) it is based on a depth-first search (DFS) of the state space of a concurrent system and (2) it uses a reduction function based on the process structure of the system. For the full details of the algorithm, the reader is referred to the original references [8,14]. In this paper, we concentrate on the condition that prevents action ignoring.

The latter is based on the fact that the DFS version of the algorithm in Fig. 1 uses a *stack* to store the unexpanded states. (Procedures that operate on the state space and the stack have the usual semantics.)

We use that observation to formulate a simple locally checkable condition that implies C2w and as such prevents action ignoring. Let $stack(s')$ be the set

```
1    Stack D = ∅
2    StateSpace V = ∅

3    Start() {
4        AddStatespace(V,ŝ)
5        PushStack(D,ŝ)
6        Search()
7    }

8    Search() {
9        s = TopStack(D)
10       for each s -a-> s' ∈ r(s)
11           if InStateSpace(V,s') == false {
12           AddStatespace(V,s')
13           PushStack(D,s')
14           Search()
15       }
16       /* cycle proviso C2 */
17       PopStack(D)
18   }
```

Fig. 1. Depth-First Search Partial-Order Reduction Algorithm

of states which are in the DFS stack D immediately before InStateSpace(V,s') is called at line 11. The new condition that is also given below is required to hold at line 16 of the algorithm in Fig. 1 as a search invariant. (In an implementation the proviso would be checked at the same place and if it is not satisfied then the for loop at line 10 will be repeated with $r(s) = \mathit{enabled}(s)$.)

- C2s: (stack proviso) For any $s \in S_r$, there exists at least one action $a \in r(s)$ and state $s' \in S_r$ such that $s \xrightarrow{a} s'$ and s' is not on the DFS stack, i.e., $s' \not\in \mathit{stack}(s')$. Otherwise, $r(s) = \mathit{enabled}_{\mathcal{T}}(s)$.

Intuitively, the stack proviso C2s allows the execution in the reduced LTS of an action a which is enabled in s, but it is outside $r(s)$, to be postponed as long as we are sure that the action will be executed in some downstream state of the DFS. This is the case as long as not all transitions of $r(s)$ lead to states on the DFS stack, i.e., not all of them close a cycle along which a could be ignored. As by persistence a is independent with any $b \in r(s)$, a remains enabled (in $\mathcal{T}$) in all states obtained from s via actions/transitions from $r(s)$. If all the transitions of $r(s)$ close a cycle, this is a potential danger that a could be ignored. To prevent this we execute all enabled transitions in s, i.e., $r(s) = \mathit{enabled}(s)$. Analogously with the discussion about the intuition behind C2w, we can conclude that because we work with finite state spaces a transition is not postponed forever, i.e., DFS will eventually hit a state in which $r(s) = \mathit{enabled}(s)$.

Formally, the correctness of the proviso C2s is implied by the following lemma:

Lemma 1. *Let $\mathcal{T}$ be an LTS and $\mathcal{T}_r$ its reduction obtained using the DFS POR algorithm in Fig. 1 with a reduction function r that satisfies condition C2s. Then r satisfies the ignoring prevention condition C2w, i.e., there exists for r a witness function $W : S_r \to \mathbb{N}$.*

Proof. Let $W : S_r \to \mathbb{N}$ be a function that enumerates the states of the reduced LTS $\mathcal{T}_r$ in the order they are removed from the DFS stack D at line 17: the state which is removed first is mapped to 0, the one which is removed last to $\mid S_r \mid -1$. If $r(s) \neq \mathit{enabled}(s)$, by proviso C2s, there exists an action $a \in r(s)$ and a state $s' \in S_r$ such that $s \xrightarrow{a} s'$ and s' is not in $\mathit{stack}(s')$. Hence, as s' is added to D later than s, it will be removed before s. Thus, we get $W(s') < W(s)$ which means that W is a witness for r. □

3 A Breadth-First Search Partial-Order Reduction Algorithm

In this section we describe Spin's BFS algorithm with an emphasis on the new version of the cycle proviso. The pseudo-code of the BFS POR algorithm is given in Fig. 2. (Procedures that operate on the state space and the queue have the usual semantics.) Note that the cycle provisos we give below are required to hold before the recursive call of Search() in line 16 of this algorithm.

```
1    Queue D = ∅
2    StateSpace V = ∅

3    Start() {
4        AddStatespace(V,ŝ)
5        AddQueue(D,ŝ)
6        Search()
7    }

8    Search() {
9        s = DelQueue(D)
10       for each s -a-> s' ∈ r(s)
11           if InStateSpace(V,s') == false {
12               AddStatespace(V,s')
13               AddQueue(D,s')
14           }
15       /* cycle proviso C2 */
16       if D != ∅ Search()
17   }
```

Fig. 2. Breadth-First Search Partial-Order Reduction Algorithm

The conditions that ensure persistence of r, C0a, C0b and C1, do not depend on the search order. Consequently, they may remain the same as in the DFS POR

algorithm. Only the condition for ignoring prevention should be changed because in BFS we can no longer count on the DFS stack. Let $visited(s')$ be the value of V immediately before the call of InStateSpace(V,s') at line 11 of the algorithm in Fig. 2. To avoid the ignoring problem in the initial BFS POR algorithm of Spin the following condition (proviso) was used.

- C2v (visited proviso): For any state $s \in S_r$ there exists at least one action $a \in r(s)$ and a state $s' \in S_r$ such that $s \xrightarrow{a} s'$ and s' has not already been visited by the BFS, i.e., $s' \not\in visited(s')$. Otherwise, $r(s) = enabled_{\mathcal{T}}(s)$.

Similarly as in the DFS case, because of the persistence of $r(s)$, the enabled transitions outside $r(s)$ remain $\mathcal{T}$-enabled in any state s' generated from s via an action in $r(s)$. If s' is a new state, it is placed in the BFS queue in order to be expanded later by the BFS. As a consequence, the enabled actions of s which are not in $r(s)$ can be postponed to be executed later in that state or in some of its descendants. If all the states generated by actions from $r(s)$ have been already visited, then we cannot guarantee anymore that some $\mathcal{T}$-enabled actions are ignored. In terms of cycles, we cannot be sure that not all actions/transitions close a cycle along which an enabled transition is "forgotten". Therefore, to be on the safe side, for such states, we include all $\mathcal{T}$-enabled actions in $r(s)$. The role of the visited states as a potentially "dangerous destination" for the actions from $r(s)$ is analogous to the one of the states on the DFS stack in the DFS case. (The formal proof that C2v implies C2w is virtually the same as the proof of Lemma 2 below.)

Inspired by [1] we give an improved version of C2v. Our algorithm can be seen as an explicit state version of the BFS POR algorithm of [1] which targets BFS in the context of symbolic model checking. Let $queue(s')$ be the value of D before the call of InStateSpace(V,s') at line 11 of the algorithm in Fig. 2. The reduction function $r(s)$, besides conditions C0a, C0b and C1, has to satisfy also the condition below.

- C2vq (visited+queue proviso): For any state $s \in S_r$ there exists at least one action $a \in r(s)$ and a state $s' \in S_r$ such that
  - $s \xrightarrow{a} s'$ and s' has not already been visited by the BFS, i.e., $s' \not\in visited(s')$, or
  - s' is in the BFS queue, i.e., $s' \in queue(s')$.

  Otherwise, $r(s) = enabled_{\mathcal{T}}(s)$.

C2vq is a refinement of C2v which excludes part of the visited states, more precisely, the ones which are in the BFS queue. The crucial point in the intuition behind C2v was that the new states (and conceptually, the $\mathcal{T}$-enabled transitions outside the ample set) were placed in the BFS queue in order to be dealt with later. But the same reasoning applies also to all the states in the BFS queue. All of them will be considered later by the BFS and one can postpone the problem of the execution of the temporarily ignored actions until they are fetched from the queue.

Thus, we can relax the requirement from C2v that the generated state must be a new one by allowing that it is visited provided that it is still in the BFS queue. The weaker proviso C2vq increases the chance to find an $r(s)$ that is a proper subset of the enabled transitions in s and to improve in this way the efficiency of the reduction.

Taking into account that the newly generated state s' (in the text of C2vq) is added to the BFS queue (line 13 of the algorithm in Fig. 2) before the ignoring proviso is checked (line 15), we define $queue'(s)$ as the value of D at line 15, i.e., immediately before the check of the proviso. As a result we obtain the following more compact version of the proviso:

- C2q (queue proviso): There exists at least one action $a \in r(s)$ and a state $s' \in S$ such that $s \xrightarrow{a} s'$ and s' is in the BFS queue, i.e., $s' \in queue'(s)$. Otherwise, $r(s) = enabled_{\mathcal{T}}(s)$.

There is an intriguing duality between the DFS stack and the BFS queue in C2s and C2q. In the DFS version it is required that the states *are not* on the DFS stack, while in the BFS case they *must be* in the BFS queue. Considering that there is nothing in the nature of the stack and the queue as data structures that may indicate such a duality, this is a rather surprising observation. It might be interesting to investigate if this kind of duality occurs also in other model checking or graph search algorithms in general.

We show below that C2vq (C2q) implies that the prevention ignoring condition C2 is satisfied too by the reduced state space, which further entails (via Theorem 2) preservation of safety properties by the BFS algorithm.

Lemma 2. *Let $\mathcal{T}$ be an LTS and $\mathcal{T}_r$ its reduction obtained using the BFS POR algorithm with a reduction function r satisfying condition C2q (C2vq). Then r satisfies the ignoring prevention condition C2w, i.e., there exists for r a witness function $W : S_r \to \mathbb{N}$.*

Proof. Let $W : S_r \to \mathbb{N}$ be a function that enumerates the states of the reduced state space in a reverse order they are added to the BFS queue D at line 13, i.e., the initial state $\hat{s}$ which is added first is mapped to $\mid S_r \mid -1$, while the state which is added last is mapped to 0. If during the BFS POR search in a given state $s \in S_r$ C2q holds for $r(s)$, this implies that there exists at least one action $a \in r(s)$, such that $s \xrightarrow{a} s'$ and s' is in the BFS queue. As s has already been removed from the BFS queue at line 9, the first-in-first-out queue policy implies that s has been added to the queue before s' (which is still in the queue). Hence, $W(s') < W(s)$ and therefore W is a witness for r. □

The correctness of the BFS POR algorithm follows by Lemma 2 and further by Theorem 2.

3.1 A BFS Queue Based Proviso for Liveness Properties

The proviso C2q can be adapted for preservation of liveness properties. It is well known (e.g. [2]) that to preserve LTL_{-X} (and with some additional restrictions on $r(s)$ also CTL^*_{-X} [3,15]) the following condition is sufficient:

– C2l (liveness cycle proviso): For any *cycle* $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} s_n = s_0$ of length $n \in \mathbb{N} \setminus \{0\}$ in $\mathcal{T}_r$, there is an $i \in \mathbb{N}$ with $0 \leq i < n$ such that $r(s_i) = enabled(s_i)$.

Unlike the safety cycle proviso C2q (which required that an action a is not ignored by at least one cycle along which it is constantly enabled in the original LTS), the liveness cycle proviso ensures that along each cycle of the reduced LTS no action is ignored. This is because at least in one state of each cycle all enabled actions are included in $r(s)$. Thus, using similar arguments as in the case of safety properties one can conclude that all actions that might have been ignored along the cycle are executed in the reduced state space.

One can ensure the validity of C2l with the following strengthened version of C2q

– C2ql: For all actions $a \in ample(s)$ and states $s' \in S$ such that $s \xrightarrow{a} s'$, s' is in the BFS queue.

The intuition behind the liveness queue proviso C2ql is more or less the same as for C2q - we do not have to worry about "losing" an ignored transition as long as the problem is delegated to the states of the queue which will be explored later. Only, unlike in the safety case, there is a stronger requirement that the ignoring is avoided along every cycle. However, like in the safety case, there is a duality between the stack based liveness proviso for DFS [14,8] and C2ql.

Lemma 3. *Proviso C2ql implies the liveness cycle proviso C2l.*

Proof. Let $\mathcal{T}_r$ be obtained using $r(s)$ which satisfies C2ql. As in the proof of Lemma 2, let us assume a witness function W which enumerates the states of S_r in the reverse order they are entered in the BFS queue. It is obvious that for each cycle $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} s_n = s_0$ $(n > 0)$ in $\mathcal{T}_r$ there exists some $0 \leq j < n$ such that $W(s_j) < W(s_{j+1})$. Before being expanded by the BFS, the state s_j is removed from the BFS queue. Thus, for any state s in the BFS queue $W(s_j) > W(s)$. Therefore, s_{j+1} is not in the queue and by C2ql $r(s_j) = enabled(s_j)$, which proves our claim. □

Unfortunately, efficient cycle detection with BFS remains an open problem. Thus, presently the practical significance of the liveness queue proviso remains to be shown.

Using the concept of state history function and Lemma 2.3 from [1] one can generalize in a quite straightforward way C2ql beyond DFS and BFS - for an arbitrary exploration order that satisfies certain conditions.

4 Experiments

We implemented the BFS POR algorithm with the queue proviso in Spin version 4.2.0. The prototype implementation was tested on examples from the Spin

Table 1. Experimental Results with Spin's Test Suite

	BFS POR with C2v			BFS POR with C2q			DFS POR with C2s		
model	states	trans	time [s]	states	trans	time [s]	states	trans	time [s]
eratosthenes	18799	42361	0.24	3205	3683	0.01	2093	2571	0.01
leader	109	109	< 0.01	109	109	< 0.01	97	97	< 0.01
leader2	16094	16332	0.23	16094	16332	0.19	14122	14241	0.09
mobile1	66389	123076	0.47	25894	36576	0.15	9971	20246	0.11
mobile2	13924	25719	0.07	6932	9819	0.03	3301	6531	0.06
petersonN (N=2)	164	290	< 0.01	135	180	< 0.01	133	171	< 0.01
petersonN (N=3)	26373	47398	0.09	13650	21135	0.03	16720	30322	0.03
petersonN (N=4)	7.16 M	27.4 M	46.12	3.65 M	7.42 M	14.0	3.19 M	6.85 M	9.63
pftp	137897	292283	0.99	61765	80416	0.35	47356	64970	0.21

distribution. The results of the experiments are shown in Table 1. The columns correspond to the original BFS POR algorithm in Spin (with the C2v cycle proviso) the new algorithm with the improved proviso C2q, and the standard DFS algorithm (which uses the C2s proviso), respectively.

For all examples (except for the leader election protocol models leader and leader2 where the results were the same) there was an improvement in the reduction compared to the old version of the algorithm with the C2v proviso. Most of the time the improvement was significant. Often the algorithm with C2q produced a reduced state space which was two to three times smaller than the one obtained with C2v. In the best case (eratosthenes) the factor was greater than five. Besides, the verification times for the improved version were better except in the cases when the space reduction was the same by both versions of BFS POR (the leader election examples). Thus, one can conclude that the implementation of condition C2q does not incur significant time overhead.

In general, the DFS and BFS reduction are incomparable regarding their efficiency. There are examples when BFS shows better performance and vice versa [1]. In our experiments though we found only one example, Peterson's mutual exclusion protocol with three processes, for which the BFS POR algorithm produced a smaller LTS (fewer states and fewer transitions) than the DFS version. In practice DFS tends to produce smaller state spaces than BFS. A possible explanation could be that on average the set of states which are on the DFS stack and which are "dangerous destination" for the cycle proviso for the DFS case is smaller than its BFS analogue - the set of all visited states minus the states in the BFS queue.

In our experiments the number of states and transitions, as well as the verification times, obtained with the improved BFS and DFS were comparable. Memory use, which is not given in Table 1, tends to be lower for DFS than for BFS.

5 Conclusions

We presented an improvement of the BFS POR algorithm implemented in Spin. The main idea behind the improvement was a modification of the so-called cycle

proviso which prevents action ignoring. Although our algorithm targets safety properties (in particular, properties expressed as Promela assertions), we also gave a strengthening of the proviso which preserves liveness properties expressed in LTL_{-X} and $CTL^*_{-}X$. The algorithm and proviso presented in the paper is independent of Spin's implementation and are compatible with any BFS exploration algorithm, thus as such they can be used in other state space exploration tools.

It would be interesting to check if our algorithm can be used in combined searches, like the combination of DFS and BFS for directed model checking as described in [12].

References

1. R. Alur, R.K. Brayton, T.A. Henzinger, S. Qadeer, and S.K. Rajamani, *Partial-order reduction in symbolic state-space exploration*, *Formal Methods in System Design*, 18:97-116, 2001. A preliminary version appeared in Proc. of the 9th International Conference on Computer-aided Verification, CAV '97, LNCS 1254, pp. 340–351, Springer, 1997.
2. E. Clarke, O. Grumberg, D.A. Peled, *Model Checking* MIT Press, 2000.
3. R. Gerth, R. Kuiper, D. Peled, W. Penczek, *A Partial Order Approach to Branching Time Logic Model Checking*, Information and Computation 150(2): 132-152, 1999.
4. P. Godefroid, *Partial Order Methods for the Verification of Concurrent Systems: An Approach to the State Space Explosion*, LNCS 1032, Springer, 1996.
5. P. Godefroid, P. Wolper, *Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties*, Computer Added Verification, CAV '91, LNCS 575, pp. 332-342, Springer, 1991.
6. G.J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*, Addison Wesley, 2003.
7. G.J. Holzmann, P. Godefroid, D. Pirottin, *Coverage Preserving Reduction Strategies for Reachability Analysis*, in Proc. 12th IFIP WG 6.1. International Symposium on Protocol Specification, Testing, and Validation, FORTE/PSTV '92, pp.349-363, North-Holland, 1992.
8. G. Holzmann, D. Peled, *An Improvement in Formal Verification*, FORTE 1994, Bern, Switzerland, 1994.
9. G. Holzmann, D. Peled, M. Yannakakis, *On Nested Depth First Search*, Proc. of the 2nd Spin Workshop, Rutgers University, New Jersay, USA, 1996.
10. S. Katz, D. Peled, *Verification of Distributed Programs Using Representative Interleaving Sequences*, Distributed Computing, 6:107-120, 1992.
11. R.P. Kurshan, V. Levin, M. Minea, D. Peled, H. Yenigün, *Static Partial Order Reduction*, in Tools and Algorithms for Construction and Analysis of Systems TACAS '98, LNCS 1384, pp. 345-357, 1998.
12. A. Lluch-Lafuente, S. Edelkamp, S. Leue, *Partial Order Reduction in Directed Model Checking*, In 9th Int. SPIN Workshop, SPIN 2002, LNCS 2318, pp. 112-127, Springer, 2002.
13. W.T. Overman, Verification of Concurrent Systems: Function and Timing, Ph.D. Thesis, UCLA, Los Angeles, California, 1981.
14. D.A. Peled, *Combining Partial Order Reductions with On-the-Fly Model Checking*, Formal Methods on Systems Design, 8: 39-64, 1996. A previous version appeared in Computer Aided Verification 1994, LCNS 818, pp. 377-390, 1994.

15. B. Willems, P. Wolper, *Partial Order Models for Model Checking: From Linear to Branching Time*, Proc. of 11 Symposium of Logics in Computer Science, LICS 96, New Brunswick, pp. 294-303, 1996.
16. A. Valmari, *Eliminating Redundant Interleavings during Concurrent Program Verification*, Proc. of Parallel Architectures and Languages Europe '89, vol. 2, LNCS 366, pp. 89-103, Springer, 1989.
17. A. Valmari, *A Stubborn Attack on State Explosion*, in Advances in Petri Nets, LNCS 531, pp. 156-165, Springer, 1991.
18. A. Valmari, *The State Explosion Problem*, Lectures on Petri Nets I: Basic Models, LNCS Tutorials, LNCS 1491, pp. 429-528, Springer, 1998.

6 Appendix

6.1 Implementation of the BFS POR Algorithm in Spin

The pseudo-code of the BFS POR algorithm implementation in Spin is given in Fig. 3. The algorithm is an extension of the iterative version of the BFS POR algorithm in Fig. 2. It is obtained by removing in a standard way the tail recursive call of Search() at line 16 in Fig. 2 and implementing the cycle proviso of line 15 in Fig. 2. To implement the cycle proviso C2q boolean variable ProvisoOK and procedure InQueue(D,s'), which checks if state s' is in the queue D, are introduced.

Another important difference with the algorithm in Fig. 2 is the while loop between lines 6 and 17. In Spin $r(s)$ is implemented via so-called ample sets. Ample sets are actually persistent sets that satisfy the cycle proviso. They consist of transitions of only one process P. This loop iterates until a candidate action set which does not contain all enabled transitions is found (indicated by ProvisoOK == true) or there are no more candidate processes that can produce an ample set. In the implementation the choice of a candidate ample set is done such that that all processes are scanned in order to find those that in its current location contain only so-called safe actions. (The safe actions are defined such that they ensure that the candidate ample set is persistent(c.f. [8])) The locations/actions are labeled as safe statically, during the scanning of the Promela model. Thus, they do not incur additional time overhead during the verification.

As it was mentioned above, the boolean variable ProvisoOK indicates if the proviso is satisfied. Before expanding each state ProvisoOK is set to *false* (line 8). In case a state is generated which is in the queue, then it is set to `true` (lines 14 and 15). If the choice of a process for an ample set was not successful (which is checked in line 18) then the ample set consists of all enabled transitions (from all processes) in state s and consequently s is correspondingly expanded (lines 18-24).

```
1   Queue D = ∅
2   StateSpace V = ∅

3   while D != ∅ {
4       s = DelQueue(D)
5       ProvisoOK = false
6       while ProvisoOK == false && there are candidate processes {
7           choose a process P for the ample set, i.e., r(s)
8           ProvisoOK = false
9           for each s -a-> s' ∈ r(s) {
10              if InStateSpace(V,s') == false {
11                  AddStatespace(V,s')
12                  AddQueue(D,s')
13              }
14              if InQueue(D,s') == true
15                  ProvisoOK = true
16          } /* transitions */
17      } /* candidate processes */

18      if ProvisoOK == false {
19          for each s -a-> s'
20              if InStateSpace(V,s') == false {
21                  AddStatespace(V,s')
22                  AddQueue(D,s')
23              }
24      }
25  }
```

Fig. 3. Pseudo-code of the implementation of BFS POR in Spin

Sound Transaction-Based Reduction Without Cycle Detection

Vladimir Levin[1], Robert Palmer[2], Shaz Qadeer[3], and Sriram K. Rajamani[3]

[1] Microsoft
vladlev@microsoft.com
[2] University of Utah
rpalmer@cs.utah.edu
[3] Microsoft Research
{qadeer, sriram}@microsoft.com

Abstract. Partial-order reduction is widely used to alleviate state-space explosion in model checkers for concurrent programs. Traditional approaches to partial-order reduction are based on *ample* sets. Natural ample sets can be computed for threads that communicate with each other predominantly through message queues. For threads that communicate with shared memory using locks for synchronization, Lipton's theory of reduction provides a promising way to aggregate several fine-grained transitions into larger transactions. In traditional partial-order reduction, actions that are not in the ample set are delayed, thus avoiding the redundant exploration of equivalent interleaving orders. Delaying the execution of actions indefinitely can lead to loss of soundness. This is called the *ignoring problem*. The usual solution to the ignoring problem is by Cycle Detection. Explicit state model checkers usually use Depth First Search, and when a cycle is detected, disallow using a reduced ample set that closes the cycle.

The ignoring problem exists in transaction-based reduction as well. We present a novel solution to the ignoring problem in the context of transaction-based reduction. We designate certain states as *commit points* and track the exploration to discover whether the reduced exploration guarantees a path from each commit point to a state where the transaction is completed. If such a path does not exist, we detect this at the time a commit point is popped from the stack, and schedule all threads at the commit point. This paper presents our algorithm, called Commit Point Completion (CPC). We have implemented both CPC and Cycle Detection in the Zing model checker, and find that the CPC algorithm performs better.

1 Introduction

Partial-order methods have been widely used as an optimization in building model checkers for concurrent software [1–6]. Traditional partial-order reduction methods are based on the notion of *independence* between actions. Two actions α and β are independent if (1) they do not disable one another and (2) if both actions are enabled in a state s, then executing them in the order α followed by

P. Godefroid (Ed.): SPIN 2005, LNCS 3639, pp. 106–122, 2005.

β from s, or in the order β followed by α from s, leads to the same resulting state. Partial-order reduction algorithms explore a subset of enabled actions in each state called the *ample* set. The set of all actions enabled in a state s is denoted $Enabled(s)$ and the ample set of actions in a state s is denoted $Ample(s)$. Obviously, $Ample(s) \subseteq Enabled(s)$. For partial-order reduction to be sound, ample sets need to be chosen in such a way that a transition that is dependent on a transition in $Ample(s)$ cannot execute without a transition in $Ample(s)$ occurring first (see condition **C1** in [1] page 148). Choosing a minimal ample set satisfying **C1** is a very hard problem. In practice, ample sets are formed from local actions, and from restricted versions of send and receive actions, such as: sending to a queue, with the sender having exclusive rights of sending to the queue, and receiving from a queue, with the receiver having exclusive rights of receiving from the queue [3]. If the system consists of threads interacting via shared memory, Lipton's theory of reduction [7] provides an alternate way to do partial-order reduction. Reduction views a transaction as a sequence of actions $a_1, \ldots, a_m, x, b_1, \ldots, b_n$ such that each a_i is a right mover and each b_i is a left mover. A right mover is an action that commutes to the right of every action by another thread; a left mover is an action that commutes to the left of every action by another thread. Thus, to detect transactions we need to detect right and left movers. Most programs consistently use mutexes to protect accesses to shared variables, we can exploit this programming discipline to infer left and right movers:

- The action `acquire(m)`, where `m` is a mutex, is a right mover.
- The action `release(m)` is a left mover.
- An action that accesses only a local variable or shared variable that is consistently protected by a mutex is both a left mover and a right mover.

A transaction is a sequence of right movers, followed by a *committing* action that is not a right mover, followed by a sequence of left movers. A transaction can be in two states: pre-commit or post-commit. A transaction starts in the pre-commit state and stays in the pre-commit state as long as right movers are being executed. When the committing action is executed, the transaction moves to the post-commit state. The transaction stays in the post-commit state as long as left movers are being executed until the transaction completes. In addition to being able to exploit programmer-imposed discipline such as protecting each shared variable consistently with the same lock, transaction-based reduction allows extra optimizations such as summarization [8].

Ignoring Problem. All partial-order reduction algorithms work by delaying the execution of certain actions, thus avoiding the redundant exploration of equivalent executions. For instance, if thread t_1 executes an action from state s_1 that reads and writes only local variables, then thread t_2 does not need to be scheduled to execute in s_1, and t_2's scheduling can be delayed without losing soundness. For any interleaving that starts from s_1 and ends in a state where some thread t goes wrong, there exists an equivalent interleaving where the execution of t_2 is delayed at s_1. However, unless we are careful, the scheduling of thread t_2 can be delayed indefinitely resulting in loss of soundness. This situation is called the *ignoring problem* in partial-order reduction.

```
int g = 0;

void T1() {                    void T2() {
L0:    g = 1;                  M0:   assert(g == 0);
L1:    skip;                   M1:   return;
L2:    while(true){            }
L3:      skip;
       }
L4:   return;                  P = { T1() } || { T2() }
}
```

Fig. 1. Ignoring problem

Consider the example in Figure 1. The initial state of this program has two threads t_1 and t_2 starting to execute functions `T1` and `T2` respectively. The program has one global variable `g`, which has an initial value `0`. A typical model checking algorithm first schedules t_1 to execute the statement at line `L0`, which updates the value of `g` to `1`. Let us call this state s_1. Since the next statement executed by thread t_1 from s_1 reads and writes only local variables of t_1 (namely its program counter) and does not read or write the global variables, transaction-based reduction algorithms delay execution of thread t_2 at state s_1. Continuing, the while loop in lines `L2` and `L3` also reads and writes only the local variables of t_1 and thus execution of t_2 can be delayed during the execution of these statements as well. However, since reached states are stored, and a newly generated state is not re-explored if it is already present in the set of reached states, a fix-point is reached after executing the loop in T_1 once. Thus, the execution of t_2 is delayed indefinitely, and the reduction algorithm can be unsound, and say that the assertion in line `M0` is never violated.

Most partial-order reduction algorithms "fix" the ignoring problem by detecting cycles, and disallowing the actions of a thread to be ample when a cycle is "closed" (see condition **C3**, pages 150 and 158 in [1]). Since explicit-state model checkers usually use depth first search (DFS), cycle detection can be performed by detecting whether a newly generated state is already present in the DFS stack. In the SPIN model checker this is implemented using a bit in the hash table entry for reached states. This bit indicates whether the newly generated successor state is currently also on the depth first search stack.

Cycle detection is neither necessary nor sufficient for transaction-based reduction. Consider the variant of our current example in Figure 2. Here, we have introduced a nondeterministic choice in line `L2` of procedure `T1`. In one branch of the nondeterministic choice, we have a while-loop with statements reading and writing only local variables of thread t_1 (lines `L3`-`L4`). The other branch of the nondeterministic choice just terminates the procedure. In this case, even without doing any cycle detection, since one branch of the nondeterministic choice terminates, a partial-order reduction algorithm can schedule thread t_2 after procedure `T1` terminates, and thus the assertion violation in line `M0` can be detected. If we consider a variant of this example, where the entire "if" statement (from

```
int g = 0;

void T1() {                    void T2() {
L0:    g = 1;                  M0:   assert(g == 0);
L1:    skip;                   M1:   return;
L2:    if (*) {                }
L3:        while(true){
L4:           skip;
L5:        }                   P = { T1() } || { T2() }
       }
L6:    return;
}
```

Fig. 2. Cycle detection is not necessary for transaction-based reduction

line L2 to L6 is replaced by `assume(false)` (see Appendix C) at line L2, some other mechanism in addition to cycle detection is needed to schedule the thread t_2 after t_1 executes the statement L1.

In the current literature on transaction-based reduction, the ignoring problem is addressed indirectly by disallowing certain types of infinite executions, such as those consisting of only internal hidden actions, within each thread (see Condition **C** from Section 4.2 in [9] which forbids the transaction from having infinite executions after committing, but without completing, and well-formedness assumption **Wf-ifinite-invis** from Section 4 in [10]). These assumptions do not hold in practice. In particular, when we analyze models that arise from abstractions (such as predicate abstraction) of programs, it is common to have loops with non-deterministic termination conditions, which violate the above assumptions. Thus, a more direct and computationally effective solution to the ignoring problem is required for wide applicability of transaction-based reduction. This paper presents a novel solution to this problem.

CPC Algorithm. We propose a new technique called *Commit Point Completion* (CPC) to solve the ignoring problem without cycle detection. We keep track of the state immediately after the committing action is executed, called the *commit point*. When a committed transaction completes, we simply mark the commit point as completed. When an unmarked commit point is about to be popped from the DFS stack, we schedule all threads from that state. Our insight is that we can delay the decision to forcibly end a transaction up to the time when commit point is about to be popped from the stack, avoiding taking such a decision pre-maturely when cycles are closed.

In the example from Figure 1 the state immediately after t_1 executes the statement at line L0 is a commit point. Due to the non-terminating while loop, the transaction that is committed here never completes. Thus, when this commit point is about to the popped from the DFS stack, it is unmarked, and the CPC algorithm schedules thread t_2 from this state, and the assertion violation in line M0 is detected. The example from Figure 2, has an identical commit point. However, since one nondeterministic branch completes the transaction, the commit point gets marked. Thus, when the commit point gets popped from the

```
Mutex m;
int x = 0; /* all accesses to x will be guarded by m*/
int y = 0; /* accesses to y are not guarded */

void T1() {                    void T2() {                    void T3() {
L0: acq(m);                    M0:  acq(m);                   N0: y = 10;
L1: y := 42;                   M1:  assert(x == 0);           }
L2: x := 1;                    M2:  rel(m);
L3: rel(m);                    }
L4: while (true)
      { skip; }
}
P = { T1() } || { T2() } || { T3() }
```

Fig. 3. CPC algorithm in the presence of left movers

DFS stack, the other thread t_2 is not scheduled. Note that the assertion failure at `M0` is detected even without scheduling thread t_2 from the commit point, because t_2 will be scheduled by the reduction algorithm after the transaction in t_1 completes on one of the nondeterministic branches.

The above description of the CPC algorithm is simplistic. In the presence of left movers there may be more than one commit point for a transaction, and all of these commit points need to reach a state where the transaction completes to ensure sound reduction. For example, consider the example shown in Figure 3. In this example, there are two global variables `x` and `y` and one mutex `m`. All accesses to `x` are protected by mutex `m`, and are thus both movers. Accesses to `y` are unprotected, and are hence non-movers. Acquires of mutex `m` are right movers and releases are left movers as mentioned earlier. Thus, when thread `T1` executes the assignment to `y` at label `L1`, its transaction commits, since the access to `y` is a non-mover. The resulting state, where `y` has just been assigned `42` and the program counter of the thread `T1` is at `L2` is a commit point. Due to the infinite while-loop at at `L4` this committed transaction never completes, and the CPC algorithm can schedule threads at the above commit point when it is about to be popped from the stack. However, for us to detect the assertion violation at line `M1` of thread `T2`, another commit point needs to be established in `T1` after the assignment to `x` at line `L2`. We handle this case by designating every state in a committed-transaction obtained by executing a "pure" left mover (i.e, a transaction that is a left mover but not a both-mover) as a commit point. Thus, in `T1`, the state after executing the release at line `L3` is also designated as a commit point, and the algorithm schedules `T2` when this state is about to be popped, leading to the assertion violation.

We have implemented the CPC algorithm in the Zing model checker at MSR. Section 5 presents experimental results that compare the CPC algorithm with a Cycle Detection algorithm for various Zing programs. The results clearly demonstrate that the CPC algorithm generally explores far fewer states than the Cycle Detection algorithm.

Outline. The rest of the paper is organized as follows. Section 2 introduces notations for describing multithreaded programs precisely. Section 3 gives an abstract framework for sound transaction-based reduction. Section 4 presents the CPC algorithm and a statement of its correctness. This section contains the core new technical results of the paper. Section 5 presents experimental results from the implementation of the CPC algorithm in the Zing model checker. Section 6 compares the CPC algorithm with related work, and Section 7 concludes the paper.

2 Multithreaded Programs

The store of a multithreaded program is partitioned into the global store *Global* and the local store *Local* of each thread. We assume that the domains of *Local* and *Global* are finite sets. The set *Local* of local stores has a special store called *wrong*. The local store of a thread moves to *wrong* on failing an assertion and thereafter the failed thread does not make any other transitions.

$$\begin{aligned} t, u &\in \quad Tid = \{1, \ldots, n\} \\ i, j &\in Choice = \{1, 2, \ldots, m\} \\ g &\in Global \\ l &\in \quad Local \\ ls &\in Locals = Tid \rightarrow Local \\ & \quad State = Global \times Locals \end{aligned}$$

A multithreaded program (g_0, ls_0, T) consists of three components. g_0 is the initial value of the global store. ls_0 maps each thread id $t \in Tid$ to the initial local store $ls_0(t)$ of thread t. We model the behavior of the individual threads using two transition relations:

$$\begin{aligned} T_G &\subseteq Tid \times (Global \times Local) \times (Global \times Local) \\ T_L &\subseteq Tid \times Local \times Choice \times Local \end{aligned}$$

The relation T_G models system visible thread steps. The relation $T_G(t, g, l, g', l')$ holds if thread t can take a step from a state with global store g and local store l, yielding (possibly modified) stores g' and l'. The relation T_G has the property that for any t, g, l, there is at most one g' and l' such that $T_G(t, g, l, g', l')$. We use functional notation and say that $(g', l') = T_G(t, g, l)$ if $T_G(t, g, l, g', l')$. Note that in the functional notation, T_G is a partial function from $Tid \times (Global \times Local)$ to $(Global \times Local)$. The relation T_L models thread local thread steps. The relation $T_L(t, l, i, l')$ holds if thread t can move its local store from l to l' on choice i. The nondeterminism in the behavior of a thread is captured by T_L. This relation has the property that for any t, l, i, there is a unique l' such that $T_L(t, l, i, l')$. In addition only T_G or T_L may be enabled at any given state. Therefore $T_G(t, g, l, g', l') \Rightarrow \forall l''\ i. \neg T_L(t, l, i, l'')$. Similarly $T_L(t, l, i, l') \Rightarrow \forall g'\ l''. \neg T_G(t, g, l, g', l'')$.

The program starts execution from the state (g_0, ls_0). At each step, any thread may make a transition. The transition relation $\rightarrow_t \subseteq State \times State$ of thread t is the disjunct of the system visible and thread local transition relations defined below. For any function h from A to B, $a \in A$ and $b \in B$, we write

$h[a := b]$ to denote a new function such that $h[a := b](x)$ evaluates to $h(x)$ if $x \neq a$, and to b if $x = a$.

$$\frac{T_G(t, g, ls(t), g', l')}{(g, ls) \rightarrow_t (g', ls[t := l'])} \qquad \frac{T_L(t, ls(t), i, l')}{(g, ls) \rightarrow_t (g, ls[t := l'])}$$

The transition relation $\rightarrow \subseteq State \times State$ of the program is the disjunction of the transition relations of the various threads:

$$\rightarrow = \exists t.\ \rightarrow_t$$

3 Transactions

Transactions occur in multithreaded programs because of the presence of right and left movers. Inferring which actions of a program are right and left movers is a problem that is important but orthogonal to the contribution of this paper. In this section, we assume that right and left movers are available to us as the result of a previous analysis (see, e.g. [11]).

Let $RM, LM \subseteq T_G$ be subsets of the transition relation T_G with the following properties for all $t \neq u$:

1. If $RM(t, g_1, l_1, g_2, l_2)$ and $T_G(u, g_2, l_3, g_3, l_4)$, there is g_4 such that $T_G(u, g_1, l_3, g_4, l_4)$ and $RM(t, g_4, l_1, g_3, l_2)$.
2. If $T_G(u, g_1, l_1, g_2, l_2)$ and $RM(t, g_2, l_3, g_3, l_4)$, then for all g', l' $(T_G(t, g_1, l_3, g', l') \Rightarrow RM(t, g_1, l_3, g', l'))$.
3. If $T_G(u, g_1, l_1, g_2, l_2)$ and $LM(t, g_2, l_3, g_3, l_4)$, there is g_4 such that $LM(t, g_1, l_3, g_4, l_4)$ and $T_G(u, g_4, l_1, g_3, l_2)$.
4. If $T_G(u, g_1, l_1, g_2, l_2)$ and $LM(t, g_1, l_3, g_3, l_4)$, there is g_4 such that $LM(t, g_2, l_3, g_4, l_4)$.

The first property states that a right mover of thread t commutes to the right of a transition of a different thread u. The second property states that if a right mover of thread t is enabled in the post-state of a transition of another thread u, and thread t is enabled in the pre-state, then the transition of thread t is a right mover in the pre-state. The third property states that a left mover of thread t commutes to the left of a transition of a different thread u. The fourth property states that a left mover that is enabled in the pre-state of a transition by another thread is also enabled in the post-state.

Our analysis is parameterized by the values of RM and LM and only requires that they satisfy these four properties. The larger the relations RM and LM, the longer the transactions our analysis infers. Therefore, these relations should be as large as possible in practice.

In order to minimize the number of explored interleaving orders and to maximize reuse, we would like to **infer** transactions that are as long as possible (i.e., they are maximal with respect to a given thread). To implement this inference, we introduce in each thread a boolean local variable to keep track of the phase of that thread's transaction. Note that this instrumentation is done automatically

by our analysis, and not by the programmer. The phase variable of thread t is true if thread t is in the right mover (or pre-commit) part of the transaction; otherwise the phase variable is false. We say that the transaction *commits* when the phase variable moves from true to false. The initial value of the phase variable for each thread is *false*.

$$\begin{aligned} p, p' \in \mathit{Boolean} &= \{\mathit{false}, \mathit{true}\} \\ \ell, \ell' \in \mathit{Local}^{\#} &= \mathit{Local} \times \mathit{Boolean} \\ \mathit{ls}, \mathit{ls}' \in \mathit{Locals}^{\#} &= \mathit{Tid} \to \mathit{Local}^{\#} \\ \mathit{State}^{\#} &= \mathit{Global} \times \mathit{Locals}^{\#} \end{aligned}$$

Let $\mathit{Phase}(t, (g, \mathit{ls}))$, the phase of thread t in state (g, ls) be the second component of $\mathit{ls}(t)$.

The initial value of the global store of the instrumented program remains g_0. The initial value of the local stores changes to ls_0, where $\mathit{ls}_0(t) = \langle \mathit{ls}_0(t), \mathit{false}\rangle$ for all $t \in \mathit{Tid}$. We instrument the transition relations T_G and T_L to generate a new transition relation $T^{\#}$.

$$T^{\#} \subseteq \mathit{Tid} \times (\mathit{Global} \times \mathit{Local}^{\#}) \times \mathit{Choice} \times (\mathit{Global} \times \mathit{Local}^{\#})$$

$$T^{\#}(t, g, \langle l, p\rangle, i, g', \langle l', p'\rangle) \stackrel{\text{def}}{=} \begin{cases} \vee\ T_G(t, g, l, g', l') \wedge \\ \quad p' = (\mathit{RM}(t, g, l, g', l') \wedge (p \vee \neg \mathit{LM}(t, g, l, g', l'))) \\ \vee\ T_L(t, l, i, l') \wedge g = g' \wedge p' = p \end{cases}$$

In the definition of $T^{\#}$, the relation between p' and p reflects the intuition that if p is true, then p' continues to be true as long as it executes right mover transitions. The phase changes to false as soon as the thread executes a transition that is not a right mover. Thereafter, it remains false as long as the thread executes left movers. Then, it becomes true again as soon as the thread executes a transition that is a right mover and not a left mover. A transition from T_L does not change the phase. We overload the transition relation $\to_t$ defined in Section 2 to represent transitions in the instrumented transition relation. Similar to the functional notation defined for T_G in Section 2, we sometimes use functional notation for $T^{\#}$.

Given an instrumented transition relation $T^{\#}$, we define three sets for each thread t: $\mathcal{R}(t), \mathcal{L}(t), \mathcal{N}(t) \subseteq \mathit{State}^{\#}$. These sets respectively define when a thread is executing in the right mover part of a transaction, the left mover part of a transaction, and outside any transaction. These three sets are a *partition* of $\mathit{State}^{\#}$ defined as follows:

- $\mathcal{R}(t) = \{\ (g, \mathit{ls}) \mid \exists l.\ \mathit{ls}(t) = \langle l, \mathit{true}\rangle \wedge l \notin \{\mathit{ls}_0(t), \mathit{wrong}\}\ \}$.
- $\mathcal{L}(t) = \left\{ (g, \mathit{ls}) \;\middle|\; \begin{array}{l} \exists l.\ \ \mathit{ls}(t) = \langle l, \mathit{false}\rangle \wedge l \notin \{\mathit{ls}_0(t), \mathit{wrong}\} \wedge \\ \qquad (\exists i, g', l'.\ \mathit{LM}(t, g, l, g', l') \vee T_L(t, l, i, l')) \end{array} \right\}$.
- $\mathcal{N}(t) = \mathit{State}^{\#} \setminus (\mathcal{R}(t) \cup \mathcal{L}(t))$.

The definition of $\mathcal{R}(t)$ says that thread t is in the right mover part of a transaction if and only if the local store of t is neither its initial value nor *wrong*

and the phase variable is true. The definition of $\mathcal{L}(t)$ says that thread t is in the left mover part of a transaction if and only if the local store of t is neither its initial value nor *wrong*, the phase variable is false, and there is an enabled transition that is either a left mover or thread-local. Note that since the global transition relation is deterministic, the enabled left mover is the only enabled transition that may access a global variable. Since $(\mathcal{R}(t), \mathcal{L}(t), \mathcal{N}(t))$ is a partition of $State^{\#}$, once $\mathcal{R}(t)$ and $\mathcal{L}(t)$ have been picked, the set $\mathcal{N}(t)$ is implicitly defined.

$$p = \overbrace{p_1 \rightarrow^+_{t(1)} p_2 \rightarrow^+_{t(2)} p_3 \cdots p_k \rightarrow^+_{t(k)} p_{k+1}}^{(p_2 = p_{2,1} \rightarrow_{t(2)} \cdots \rightarrow_{t(2)} p_{2,x} = p_3)} = \overbrace{q_1 \rightarrow^+_{u(1)} q_2 \rightarrow^+_{u(2)} q_3 \cdots q_l \rightarrow^+_{u(l)} q_{l+1}}^{(q_2 = q_{2,1} \rightarrow_{u(2)} \cdots \rightarrow_{u(2)} q_{2,x} = q_3)} = q$$

Fig. 4. A sequence of transactions.

A sequence of states from Figure 4 is called a *sequence of transactions* if

- for all $1 \leq m \leq k$, if $p_m = p_{m,1} \rightarrow_{t(m)} \cdots \rightarrow_{t(m)} p_{m,x} = p_{m+1}$, then (1) $p_{m,1} \in \mathcal{N}(t(m))$, (2) $p_{m,2}, \ldots, p_{m,x-1} \in \mathcal{R}(t(m)) \vee \mathcal{L}(t(m))$, and (3) $p_{m,x} \in \mathcal{L}(t(m)) \vee \mathcal{N}(t(m))$.
- for all $1 \leq m \leq l$, if $q_m = q_{m,1} \rightarrow_{u(m)} \cdots \rightarrow_{u(m)} q_{m,x} = q_{m+1}$, then (1) $q_{m,1} \in \mathcal{N}(u(m))$, and (2) $q_{m,2}, \ldots, q_{m,x} \in \mathcal{R}(u(m))$.

Intuitively, for every i, $p_i \rightarrow^+_{t(i)} p_{i+1}$ is a committed transaction and for every j, $q_j \rightarrow^+_{u(j)} q_{j+1}$ is an uncommitted transaction.

The following theorem says that for any sequence in the state space that reaches a state where some thread t goes wrong, there exists a corresponding sequence of transactions that reaches a corresponding state at which thread t goes wrong.

Theorem 1. *Let $P = (g_0, \ell s_0, T^{\#})$ be the instrumented multithreaded program. For all $t \in Tid$, let $\mathcal{W}(t) = \{(g, \ell s) \mid \exists p.\ \ell s(t) = \langle wrong, p \rangle\}$. For any state $(g', \ell s') \in \mathcal{W}(t)$ that is reachable from $(g_0, \ell s_0)$, there is another state $(g'', \ell s'') \in \mathcal{W}(t)$ that is reachable from $(g_0, \ell s_0)$ by a sequence of transactions.*

A detailed proof of this theorem can be found in our technical report [12]. As a consequence of this theorem, it suffices to explore only transactions to find errors. This is the basis for the our reduction algorithm. Using the values of $\mathcal{N}(t)$ for all $t \in Tid$, we model check the multithreaded program by computing the least fixpoint of the set of rules in Figure 5. This model checking algorithm schedules a thread only when no other thread is executing inside a transaction.

This algorithm is potentially unsound for the following reason. If a transaction in thread t commits but never finishes, the shared variables modified by this transaction become visible to other threads. However, the algorithm does not explore transitions of other threads from any state after the transaction commits. Section 4 presents a more sophisticated algorithm which ensures that all threads are explored from some state in the post-commit phase of every transaction.

$$\text{(INIT)}\quad \frac{}{\Sigma(g_0, \ell s_0)}$$

$$\text{(STEP)}\quad \frac{\forall u \neq t.\ (g, \ell s) \in \mathcal{N}(u) \quad \Sigma(g, ls) \quad T^{\#}(t, g, \ell s(t), i, g', \ell')}{\Sigma(g', \ell s[t := \ell'])}$$

Fig. 5. Model checking with unsound reduction.

4 Commit Point Completion

This section presents the CPC algorithm and its soundness theorem, which are the core new technical contributions of this paper. The algorithm uses Depth First Search (DFS). Each state in the DFS stack is encapsulated using a `TraversalInfo` record. In addition to the state, the `TraversalInfo` records the following 7 fields:

1. `tid`, the id of the thread used to to reach the state,
2. `numTids`, the number of threads active in the state,
3. `choice`, the current index among the nondeterministic choices executable by thread `tid` in this state,
4. `LM`, a boolean which is set to true iff the action used to reach this state is a left mover,
5. `RM`, a boolean which is set to true iff the action used to reach this state is a right mover,
6. `Xend`, a boolean which is set to true iff the algorithm decides to schedule other threads at this state, and
7. `CPC`, a boolean which is relevant for only states with `phase` equal to false, and is set to true by the algorithm if there exists a path of transitions of the thread generating the state to a state where all threads are scheduled.

Figure 6 gives two variants of the CPC algorithm (with and without line `L19`). The statement at `L4` peeks at the TraversalInfo `q` on top of the stack and explores all successors of the state using actions from thread `q.tid`. If the phase of `q` is false, then for each such successor `q'`, if the action used to generate `q'` is not a left-mover, then we update `q.Xend` to true at label `L7`. The invariant associated with the `CPC` flag is the following: If `q` is about to be popped from the stack and `q.CPC` is true and $Phase$(`q.tid`, `q.state`) is false then there exists a path to a state where `Xend` is true. Thus, at label `L8` we set `q.CPC` to true if `q.Xend` is true. The `Xend` and `CPC` fields are also updated when a TraversalInfo is popped from the stack. In particular, at label `L18`, when `q` is about to be popped from the stack, if its phase is false and `q.CPC` is false, then we set `q.Xend` to true and force scheduling of all threads at `q`. If `q.Xend` is true, then at label `L24` we ensure that all threads are scheduled from `q`. Figure 7 contains helper procedures for the CPC algorithm.

```
Hashtable table;
Stack stack;
TraversalInfo q, q', q'', pred;

stack = new Stack
table = new Hashtable

L0: q' = { state = (g0, ℓs0),
       tid = 1,
       numTids = 1,
       choice = 1,
       CPC = true,
       Xend = true,
       LM = false,
       RM = false }

L1: table.Add(q'.state, q')
L2: stack.Push(q')

L3: while (stack.Count > 0)
L4:   q = stack.Peek()
L5:   if (Enabled(q))
L6:     q' = Execute(q)
L7:     q.Xend = q.Xend || (¬Phase(q.tid, q.state) && ¬q'.LM)
L8:     q.CPC = q.CPC || q.Xend
L9:     if (IsMember(table, q'.state))
L10:      q'' =  Lookup(table, q'.state)
L11:      q.CPC = q.CPC || q''.CPC
L12:    else /* undiscovered state */
L13:      table.Add(q'.state, q')
L14:      stack.Push(q')
L15:    end if
L16:    q.choice = q.choice + 1
L17:  else
L18:    q.Xend = q.Xend || (¬Phase(q.tid, q.state) && ¬q.CPC
L19:                                              (* && ¬q.RM *))
L20:    q.CPC = q.CPC || q.Xend

L21:    stack.Pop()
L22:    pred = stack.Peek()
L23:    pred.CPC = pred.CPC || q.CPC

L24:    if (q.Xend && q.numTids < |Tid|)
L25:      q' = Update(q)
L26:      stack.Push(q')
L27:    end if
L28:  end if
L29:end while
```

Fig. 6. CPC algorithm for sound reduction

A key invariant preserved by the algorithm is the following: Suppose a `TraversalInfo` record `q` is about to be popped from the search stack and `q.CPC` is true. Then there is a sequence of left mover transitions of thread `q.tid` to a state represented in some `TraversalInfo` record `q'` such that `q'.Xend` is true. We can show this by induction on the order in which `TraversalInfo` records are popped from the stack (See our technical report [12]).

Without the optimization in line `L19`, the CPC algorithm ensures that for every `TraversalInfo` record `q` explored by the algorithm such that `q.state` is in the post-commit part of the transaction, there exists a sequence of transitions to some other state where all threads are scheduled. With the optimization in line `L19`, the CPC algorithm guarantees this property only for a subset of states in the post-commit part of the transaction that are reached by pure left movers as stated below.

Theorem 2. *Let q be a* `TraversalInfo` *constructed during the execution of the CPC algorithm such that* `q.RM` *= false. Then at line* `L21` *there exists a sequence of left-mover transitions of thread* `q.tid` *from* `q.state` *to $(g', \ell s')$ and all threads are explored from $(g', \ell s')$.*

Finally, Theorem 3 concludes that if there is a state in the multithreaded program where a thread goes wrong that is reachable from the initial state the CPC algorithm will find a state that is reachable from the initial state where that thread goes wrong.

Theorem 3. *If there is an execution of the multithreaded program from $(g_0, \ell s_0)$ to $(g, \ell s)$ and a thread t such that $\ell s(t) =$ wrong, then there is another state $(g', \ell s')$ where the CPC algorithm visits $(g', \ell s')$ and $\ell s'(t) =$ wrong.*

The proof involves using Theorem 1 to first produce a sequence of transactions that also reach a state where thread t goes wrong, and then using Theorem 2 to transform this latter sequence into another sequence that will be explored by the CPC algorithm. Details can be found in [12].

5 Experimental Results

We implemented the CPC algorithm in Zing, which is a software model checker being developed in Microsoft Research. Table 1 gives the number of states explored by Zing on various example programs using three variants of the reduction algorithm. The column labeled "Loc" gives the number of lines of code in the Zing program. The column labeled "Unsound Reduction" gives the number of states explored by a reduction algorithm which does not solve the ignoring problem. This gives a lower bound on the number of states that need to be explored by any sound algorithm. The column labeled "CPC" gives the number of states explored by the CPC algorithm. The column labeled "Cycle Detection" gives the number of states explored by a sound algorithm which forcibly ends a transaction whenever a cycle is encountered in the post-commit part of the transaction.

Table 1. Number of states visited by Unsound Reduction, CPC and Cycle Detection algorithms

Example	Loc	Unsound Reduction	CPC	Cycle Detection
`AuctionHouse`	798	108	108	108
`FlowTest`	485	4656	4656	4656
`Shipping`	1844	206	206	222
`Conc`	392	512	512	2063
`Peterson`	793	1080	1213	3427
`Bluetooth`	2768	48109	52092	116559
`TransactionManager`	6927	1220517	1264894	1268571
`AlternatingBit`	130	1180	1180	1349
`Philosophers`	76	87399	87399	428896
`Bakery`	104	10221	14935	14254

The number of states explored is a measure of the running time of the algorithm. The smaller the number of states explored by a sound algorithm, the faster the tool is.

The programs are classified into four groups. The first 3 programs, `AuctionHouse`, `FlowTest` and `Shipping` programs were produced by translating to Zing from a process co-ordination language called BPEL. They represent workflows for business processes, and have mostly acyclic state spaces. In these examples, the number of states explored by the all three algorithm are almost identical.

The next 3 programs `Conc`, `Peterson` and `Bluetooth` were produced by automatic abstraction refinement from concurrent C programs. We have adapted the SLAM toolkit [13] to concurrent programs by using Zing as a back-end model checker instead of Bebop. These examples all have loops that terminate non-deterministically in the abstraction. Thus, the cycle detection algorithm forces interleaving of all threads in these loops whereas the CPC algorithm avoids interleaving all threads in the loops without losing soundness. The CPC algorithm really shines in comparison with the Cycle Detection algorithm on these examples.

The `TransactionManager` program was obtained from a product group in Microsoft. It was automatically translated to Zing from C#, after a few manual abstractions and manually closing the environment. It is one of the larger Zing examples we currently have. Since the manual abstraction did not result in non-deterministically terminating loops, the CPC algorithm performs only marginally better than the Cycle Detection algorithm.

The final 3 programs, `AlternatingBit`, `Philosophers` and `Bakery` are standard toy examples used by the formal verification community. In the first two examples, CPC performs better than Cycle Detection. In the `Bakery` example we find that the Cycle Detection algorithm performs slightly better than the CPC algorithm. This is possible, since the total number of states is counted over all transactions, and the CPC algorithm gives optimality only within a single

transaction. Heuristically, this should translate to smaller number of states explored over all the transactions, but this example shows that this is not always the case.

Overall, the results clearly demonstrate that CPC is a good algorithm for making reduction sound, without forcing the interleaving of other threads in all loops. It generally explores fewer states than Cycle Detection, and out-performs Cycle Detection in examples with nondeterministic loops. Such examples arise commonly from automatic abstraction refinement.

6 Related Work

Partial-order reduction has numerous variants. The most commonly used ones are stubborn sets of Valmari [2], ample sets [4, 1], and sleep sets [5]. Most of these approaches handle the ignoring problem by using some variant of cycle detection. In another paper, Valmari proposes detecting Strongly Connected Components (SCCs) to solve the ignoring problem [14]. This algorithm from [14] involves detecting terminal strongly connected components, and forces scheduling of other threads from at least one state in each of the terminal strongly connected components (see Algorithm 1.28, Section 5 in [14]). In contrast, the CPC algorithm does not directly compute any strongly connected components. Also the CPC algorithm terminates transactions at fewer points than Valmari's algorithm. See Appendix B for am example.

Transaction based reduction was originally developed by Lipton [7]. Work by Stoller and Cohen [10] uses a locking discipline to aggregate transitions into a sequence of transitions that may be viewed atomically. Flanagan and Qadeer augment this approach with right movers to get further reduction [9]. This idea is combined with procedure summarization by Qadeer, Rajamani, and Rehof in [8]. As mentioned earlier, all of these papers address the ignoring problem only indirectly by disallowing certain types of infinite executions, such as those consisting of only internal hidden actions, within each thread (see Condition **C** from Section 4.2 in [9] which forbids the transaction from having infinite executions after committing, but without completing, and well-formedness assumption **Wf-ifinite-invis** from Section 4 in [10]). It is not clear how these assumptions are enforced. Two of the above papers [9, 8] do not have any accompanying implementation, and it is unclear how the ignoring problem is solved in the implementation associated with [10]. Our guess is that they use some form of cycle detection.

The Verisoft [6] implementation does not use the detection of cycles or strongly connected components, rather a timeout is used to detect an infinite execution that is local to a particular process. Other cycles are broken by limiting the search depth or using a driver that generates a finite number of external events. Dwyer et al [15] use the notion of a locking discipline is used to increase the number of transitions that can form an ample set for a process. The algorithms presented use the standard cycle detection technique to insure soundness.

7 Conclusion

Partial-order reduction methods with ample sets usually use Cycle Detection to solve the ignoring problem. In the context of transaction based reduction, we propose a new technique called Commit Point Completion (CPC) to solve the ignoring problem. We have proved that this algorithm is correct, and have implemented it in the Zing model checker. Our experimental results demonstrate that with transaction based reduction, the CPC algorithm performs better than Cycle Detection. Though the CPC algorithm was presented using the terminology of Lipton's transactions, we believe that the idea is applicable to other variants of partial-order reduction as well. Exploration of this idea is left to future work. The ignoring problem also arises when we attempt to build summaries for multithreaded programs[8]. Though not mentioned here, our implementation of summaries in Zing also uses the core idea of the CPC algorithm to ensure soundness.

References

1. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
2. Valmari, A.: A stubborn attack on state explosion. In: CAV 91: Computer Aided Verification, Springer-Verlag (1991) 156–165
3. Holzmann, G., Peled, D.: An improvement in formal verification. In: FORTE 94: Formal Description Techniques, Chapman & Hall (1994) 197–211
4. Peled, D.: Partial order reduction: Model-checking using representatives. In: MFCS 96: Mathematical Foundations of Computer Science, Springer-Verlag (1996) 93–112
5. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. LNCS 1032. Springer-Verlag (1996)
6. Godefroid, P.: Model checking for programming languages using Verisoft. In: POPL 97: Principles of Programming Languages. (1997) 174–186
7. Lipton, R.J.: Reduction: A method of proving properties of parallel programs. In: Communications of the ACM. Volume 18:12. (1975) 717–721
8. Qadeer, S., Rajamani, S.K., Rehof, J.: Summarizing procedures in concurrent programs. In: Principles of Programming Languages, ACM (2004) 245–255
9. Flanagan, C., Qadeer, S.: Transactions for software model checking. In: SoftMC 03: Software Model Checking Workshop. (2003)
10. Stoller, S.D., Cohen, E.: Optimistic synchronization-based state-space reduction. In: TACAS 03. LNCS 2619, Springer-Verlag (2003) 489–504
11. Flanagan, C., Qadeer, S.: Types for atomicity. In: TLDI 03: Types in Language Design and Implementation, ACM (2003) 1–12
12. Levin, V., Palmer, R., Qadeer, S., Rajamani, S.K.: Sound transction-based reduction without cycle detection. Technical Report MSR-TR-2005-40, Microsoft Research (2005) ftp://ftp.research.microsoft.com/pub/tr/TR-2005-40.pdf.
13. Ball, T., Rajamani, S.K.: The SLAM project: Debugging system software via static analysis. In: POPL 02: Principles of Programming Languages, ACM (2002) 1–3
14. Valmari, A.: Stubborn sets for reduced state space generation. In: Advances in Petrinets. LNCS 483, Springer-Verlag (1990)
15. Dwyer, M.B., Hatcliff, J., Robby, Ranganath, V.P.: Exploiting object excape and locking information in partial-order reducitons for concurrent object-oriented programs. Formal Methods in System Design **25** (2004) 199–240

A Helper Functions for the CPC Algorithm

```
Boolean Enabled(TraversalInfo q) {
   let (g,ℓs) = q.state in
   return (∃ g′,ℓ′. T#(q.tid, g, ℓs(q.tid), q.choice, g′, ℓ′))
}

TraversalInfo Execute(TraversalInfo q) {
   let (g,ℓs) = q.state in
   let (g′,ℓ′) = T#(q.tid, g, ℓs(q.tid), q.choice) in
   State succ = (g′, ℓs[q.tid := ℓ′])
   return { state = succ,
            tid = q.tid,
            numTids = 1,
            choice = 1,
            CPC = false,
            Xend= false,
            LM = LM(q.tid, q.state, succ)
            RM = RM(q.tid, q.state, succ) }
}

TraversalInfo Update(TraversalInfo q) {
   Tid nextTid = ite((q.tid == |Tid|), 1, q.tid + 1)
   return { state = q.state,
            tid = nextTid,
            numTids = q.numTids + 1,
            choice = 1,
            CPC = q.CPC,
            Xend= q.Xend,
            LM = q.LM,
            RM = q.RM }
}
```

Fig. 7. Helper procedures for the CPC algorithm.

The helper functions for the CPC algorithm perform the following actions. `Enabled` determines whether the current thread has a transition enabled at a given state. `Execute` applies the transition relation $T^{\#}$ to the current state. `Update` schedules the next thread to run.

B Comparison with Valmari's SCC Algorithm

Consider the example from Figure 8. In this example, a transaction commits at the state after executing line `L0`, followed by a non-deterministic branch at line

```
int g = 0;

void T1() {                          void T2() {
L0:    g = 1;                        M0:   assert(g == 0);
L1:    skip;                         M1:   return;
L2:    if (*) {                      }
L2:        while(true){
L3:           skip;
L4:        }
       }
       else {
L5:        while(true){
L6:           skip;
L7:        }
       }

L8:    return;

}

P = { T1() } || { T2() }
```

Fig. 8. Distinction between CPC algorithm and SCC-based algorithms

L2. Each of the branches produce terminal SCCs in the state space. Valmari's algorithm appears to force scheduling T2 at each of these terminal SCCs, whereas the CPC algorithm forces scheduling T2 only once, at the commit-point (label L1).

C Assume(False)

The assume(condition) construct is used in the Zing modeling language to instruct the Zing runtime to silently ignore any execution path in which the condition is found to be false. Unlike an assert statement, this is not considered to be an error condition.

Repairing Structurally Complex Data

Sarfraz Khurshid, Iván García, and Yuk Lai Suen

Department of Electrical and Computer Engineering,
The University of Texas at Austin,
1 University Station C5000,
Austin, TX 78712
{khurshid, igarcia, suen}@ece.utexas.edu

Abstract. We present a novel algorithm for repairing structurally complex data. Given an assertion that represents desired structural integrity constraints and a structure that violates them, the algorithm performs repair actions that mutate the given structure to generate a new structure that satisfies the constraints. Assertions are written as imperative predicates, which can express rich structural properties. Since these properties can be arbitrarily complex, our algorithm is sound but not complete, and it may not terminate in certain cases. Experimental results with our prototype implementation, Juzi, show that it is feasible to efficiently repair a variety of complex data structures that are routinely used in library code. Juzi can often repair structures comprising of over a hundred objects (even when majority of the objects have some corrupted field) in less than one second. Our algorithm is based on systematic backtracking but does not require storing states and can easily be implemented in a variety of software model checkers, such as the Java PathFinder, SPIN, and VeriSoft.

1 Introduction

Assertions have long been used to state crucial properties of code. A variety of tools and techniques make use of assertions to check program correctness statically at compile-time or dynamically at run-time [10, 20, 3, 4, 23, 5, 7, 12, 14, 16, 29]. If an assertion violation is detected at run-time, the program is deemed to have reached an inconsistent state. The usual process then is to terminate the execution, debug the program (if necessary and possible), and re-execute it.

In some cases, however, termination and re-execution is not feasible. And at times, it is simply impossible to re-execute a program on a desired input since the input may now represent (persistent) data that has been corrupted. Even correct programs can have corrupt data due to errors in transmission, hardware, etc. In such cases, it may be desirable to have a routine that can repair the corrupted data and bring the data in a state that is consistent with the integrity constraints expressed in the assertion, which would enable the program (to continue) to execute. A goal of such a routine is not to bring the data in a state that a correct execution/environment would have resulted in, but to bring it in a state that is acceptable to the user for continuing program execution [25].

We present a novel algorithm [27,11] for repairing *structurally complex data*, which pervade modern software, in particular object-oriented programs. A defining characteristic of such data is their *structural integrity constraints*, e.g., in a binary tree, there

P. Godefroid (Ed.): SPIN 2005, LNCS 3639, pp. 123–138, 2005.

are no cycles. Examples of complex structures include textbook data structures, such as circular linked lists and red-black trees, which are routinely used in library code to implement a variety of abstract data types. Complex structures arise in various other contexts, e.g., intentional naming systems [1] and fault-tree analyzers [28].

The integrity constraints of a structure can be written as a formula that evaluates to true if and only if the input satisfies the desired constraints. Such formulas can be written declaratively [15,23,17], e.g., using fist-order logic, or imperatively [4,22], e.g., using a Java or C++ predicate (i.e., a method that returns a boolean). Declarative notations often provide a more natural and succinct way of expressing constraints. However, these notations are usually syntactically and semantically different from common programming languages, which can impede their wide-spread adoption among practitioners.

Our repair algorithm uses imperative descriptions of constraints. In object-oriented programs such constraints are often already present as class invariants (which are usually called `repOk` methods) [21]. Given a predicate that represents desired structural integrity constraints and a structure that violates them, the algorithm performs repair actions that mutate the given structure to generate a new structure that satisfies the constraints. Each repair action assigns some value to a field of an object in the structure.

The repair actions are governed by the (1) exploration of the set of field assignments to reference variables and (2) evaluation of constraints on values of primitive data fields. Due to the enormous number of combinations of field assignments, it is not possible to simply enumerate all possible assignments (even for small structures) and check whether any assignment represents a repaired structure. For efficient repair, our algorithm employs (1) pruning techniques that are based on our previous work on the Korat framework for specification-based testing of Java programs [4], and (2) decision procedures for primitive data, similar to our previous work on test input generation using symbolic execution [18].

The algorithm executes the predicate on the corrupted structure and monitors the execution to record the order in which fields are accessed before the execution returns false. The algorithm then backtracks on the last field that was accessed and either assigns that field a different reference or assigns it a symbolic primitive value (which is different from the original value), and re-executes the predicate using (forward) symbolic execution [19] where needed. To determine the feasibility of path conditions, our prototype implementation, Juzi, uses CVC Lite [2].

At its core, our algorithm performs a systematic search using backtracking based on field accesses and on results of decision procedure invocations. Our algorithm does not require storing states. These characteristics make it very easy to implement our algorithm to work in conjunction with a variety of software model checkers, such as the Java PathFinder [29], SPIN [14], and VeriSoft [12].

Imperative predicates enable formulation of rich structural properties. Since these properties can be arbitrarily complex, our algorithm is sound but not complete: the repaired structures that the algorithm returns satisfy the constraints, but the algorithm may not terminate in certain cases. Experimental results with our prototype implementation show that it is feasible to efficiently repair a variety of complex data structures, which are used routinely in library code. Juzi can repair structures with a hundred nodes—half of which have some field that needs repair—in less than one second.

1.1 Background

Fault-tolerance and error recovery have been a part of software systems for a long time. File system utilities, such as `fsck`, routinely check and correct the underlying file structure. Some commercially developed systems, such as IBM MVS operating system [24] and the Lucent 5ESS telephone switch [13], have provided routines for monitoring and maintaining data structure properties. These routines, however, typically focus on repairing particular structures by performing specific repair actions that work only in the contexts for which they are designed.

Checkpointing and roll-back are standard mechanisms in databases to recover data to the last known good state. DIRA [26] adapts these mechanisms to detect buffer overflow attacks and repair the structures damaged by the attack.

Demsky and Rinard have recently proposed a generic model-based framework for data structure repair [9]. Given consistency constraints in a declarative language, their repair algorithm translates these constraints into a repair routine, which corrects the given corrupt structure. A distinguishing feature of our work from previous work on repair is that we provide a generic repair algorithm that does not require any input from the user beyond a description of the desired constraints and does not require learning a language different from the underlying programming language.

1.2 Contributions

This paper makes the following contributions:

- **Imperative constraints in data structure repair**. Our use of imperative constraints in the context of generic data structure repair is novel. It enables users to write constraints in a familiar notation and eliminates the need for requiring mappings between abstract models of data and concrete values; such mappings are often required when the constraint language differs from the implementation language [9].
- **Forward symbolic execution in data structure repair**. Forward symbolic execution has traditionally been used to check correctness of programs (via static or dynamic analyses) and to debug the programs. We have developed an unconventional application of symbolic execution: we use it to repair the *data* on which the programs operate on.
- **Data structure repair algorithm**. We build on algorithms from our previous work on specification-based test generation to develop a novel technique for performing generic data structure repair.
- **Repair as an application of a model checker**. We have designed our algorithm to work with off-the-shelf model checkers. To our knowledge this is the first instance that shows how a standard model-checking tool can efficiently perform data structure repair.
- **Repair studies from library code**. We perform repairs on a suite of complex structures used routinely in library code and evaluate the feasibility of structure repair. Experiments show that moderately large structures, e.g., red-black trees with a few hundred nodes, can often be repaired within a few seconds.

2 Examples

We present two examples of repairing linked structures to illustrate the use of our algorithm and prototype implementation. The first example illustrates an acyclic data structure that has been corrupted. The second example illustrates a structure that has cycles and also shows how repair can sometimes even correct program behavior on-the-fly.

2.1 Binary Tree

Consider the following declaration of a binary tree:

```
class BinaryTree {
    Node root;
    int size;

    static class Node {
        Node left;
        Node right;
    }

    boolean repOk() { ... }
}
```

Each tree has a `root` node and caches the number of nodes in the `size` field. Each node has a `left` child and a `right` child. The method `repOk` checks the structural integrity constraints for `BinaryTree`. It can be implemented as a simple graph traversal algorithm that checks for acyclicity by keeping track of the set of visited nodes and checking that it never encounters the same node twice. The method also checks for consistency of the `size` field. (Appendix A gives an implementation of `BinaryTree.repOk`.)

As an illustration of the repair algorithm, consider the corrupted structure shown in Figure 1(a). Incorrect values of fields `left` and `right` in some of the nodes result in the structure having directed cycles, which violates acyclicity. Given a description of this structure and the `repOk` method, our repair algorithm produces the repaired structure shown in Figure 1(b). Note that the field assignments now satisfy the desired constraints. To repair the corrupted structure shown in this example, Juzi takes a tenth of a second.

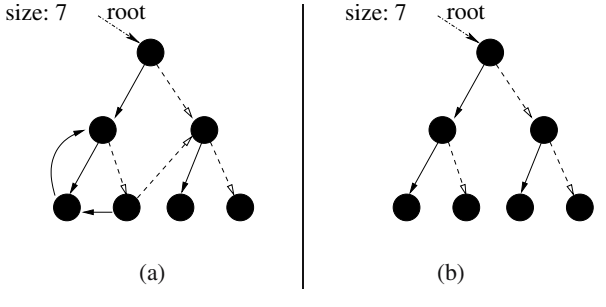


Fig. 1. Repairing a binary tree. (Solid arrows represent `left` fields; dashed arrows represent `right` fields; and `root` field is labeled appropriately.) (a) A corrupted binary tree structure: values of left and right fields of some nodes introduce directed cycles in the structure. (b) Tree resulting after repair has been performed.

2.2 Doubly-Linked List

We illustrate how repair can potentially even correct program behavior on-the-fly. The class `LinkedList` declares doubly-linked circular lists similar to those implemented in `java.util.LinkedList`:

```
class LinkedList {
    Entry header; // sentinel header entry
    int size;     // number of non-sentinel entries

    static class Entry {
        Object element;
        Entry next;
        Entry previous;
    }
}
```

The inner class `Entry` models the entries in a list. Each list has a `header` entry, which is treated as a *sentinel*. An empty list consists of just the header entry, whose next and previous fields point to itself. The `size` field stores the number of non-sentinel entries in the list.

Consider a method `LinkedList.addFirst` that given an object `o`, adds a new entry with element `o` at the head of `this` list (i.e., it makes the new entry the first non-sentinel entry in the list while preserving the original entries of the list). The following code gives an erroneous implementation of the `addFirst` method:

```
void addFirst(Object o) {
    Entry t = header.next;
    Entry e = new Entry();
    e.element = o;
    header.next = e;
    e.previous = header;
    e.next = t;
    t.previous = header;
}
```

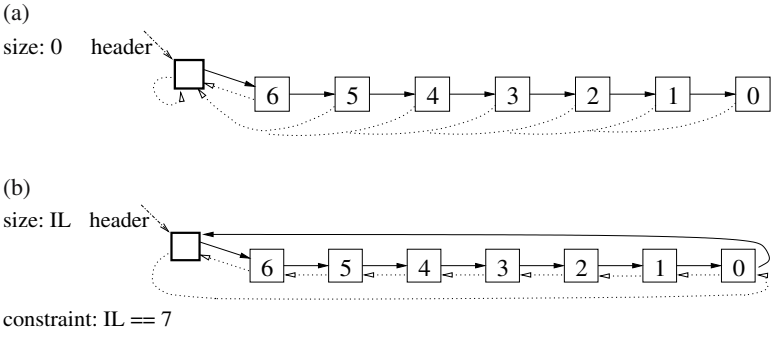


Fig. 2. Repairing a doubly-linked list. (Solid arrows represent `next` fields; dotted arrows represent `previous` fields; and `header` field that points to the sentinel node is appropriately labeled.) (a) List generated as result of erroneous `addFirst`: all `previous` pointers incorrectly point to the `header` entry, and `size` is set to 0. (b) List resulting after repair has been performed: all reference fields have correct values and `size` field is correctly constrained to 7.

The above code contains two bugs:

- it does not maintain the correspondence between `next` and `previous` fields of an entry as it erroneously sets `t.previous` to `header` instead of setting it to `e`
- it does not update the value of the `size` field.

Figure 2(a) illustrates the list that is generated by inserting integer objects with values $[0, \ldots, 6]$ in that order into an empty list using the erroneous `addFirst`. Notice the incorrect values for `previous` pointers (dotted arrows) and `size` field. All `previous` pointers incorrectly point to the `header` entry, and `size` is 0 even though there are 7 non-sentinel entries in the list.

Given this corrupted list and the `LinkedList.repOk` method, which we have not given here due to brevity, Juzi generates the structure illustrated in Figure 2(b). Notice how the `previous` pointers have been set to correct values and how the size field is constrained to have the correct value 7. For this example, Juzi took a tenth of a second to complete the repair.

3 Background: Symbolic Execution

Forward symbolic execution is a technique for executing a program on symbolic values [19]. There are two fundamental aspects of symbolic execution: (1) defining semantics of operations that are originally defined for concrete values and (2) maintaining a *path condition* for the current program path being executed—a path condition specifies necessary constraints on input variables that must be satisfied to execute the corresponding path.

As an example, consider the following program that returns the absolute value of its input:

```
       int abs(int i) {
L1.        int result;
L2.        if (i < 0)
L3.            result = -1 * i;
L4.        else result = i;
L5.        return result;
       }
```

To symbolically execute this program, we consider its behavior on a primitive integer input, say I. We make no assumptions about the value of I (except what can be deduced from the type declaration). So, when we encounter a conditional statement, we consider both possible outcomes of the condition. To perform operations on symbols, we treat them simply as variables, e.g., the statement on line 3 updates the value of `result` to be `-1 * I`. Of course, a tool for symbolic execution needs to modify the type of `result` to note updates involving symbols and to provide support for manipulating expressions, such as `-1 * I`.

Symbolic execution of the above program explores the following two paths:

```
path 1:
    [I < 0] L1 -> L2 -> L3 -> L5
path 2:
    [I >= 0] L1 -> L2 -> L4 -> L5
```

Note that for each path that is explored, there is a corresponding path condition (shown in square brackets). While execution on a concrete input would have followed exactly one of these two paths, symbolic execution explores both.

4 Algorithm

This section describes our repair algorithm. Given a structure `s` that is to be repaired and a predicate `repOk` that represents the structural constraints, the algorithm:

- invokes `s.repOk();`
- monitors execution of `repOk` to note the order in which fields of objects in `s` are accessed;
- if `repOk` returns false
  - backtracks and mutates `s` by toggling the value of the last field[1] that was accessed by `repOk` (while maintaining the values of all other fields), and re-executes `repOk`
- else
  - if (pathCondition is feasible)
    - outputs `s` (which now has been repaired)

The first execution of the algorithm is on the corrupted structure. Notice that all fields of this structure have concrete values. Therefore, the first invocation of `repOk` simply follows Java semantics. But when `repOk` returns false, the algorithm mutates the given structure, and may introduce fields that have symbolic values for primitive data; value updates to these field then follow standard forward symbolic execution [19].

To modify a field value when backtracking, the algorithm considers two primary cases[2]:

- primitive field access: the field is assigned a symbolic value `I` and the current path-condition is updated to reflect that `I != v`, where `v` is the original value of this field in the corrupt structure;
- reference field access: the field is nondeterministically assigned
  - `null`, if the original field value was not `null`;
  - an object (of a compatible type) that was encountered during the last execution of repOk on the corrupt structure, if the field was not originally pointing to this object;
  - a new object (of a compatible type), unless the object that the field originally pointed to was different from all objects in the structure encountered during the last execution of `repOk`.

It is tempting to think that a reference field of an object in a structure can potentially point to any other object that has a compatible type in that structure and to explore all such assignments. However, our algorithm does not explore them all. It turns out that it

[1] If all values for the last field accessed have already been explored, reset the value of that field to its initial value and backtrack further to modify the value of the second-last field accessed and so on.

[2] Our current prototype does not handle arrays.

suffices to select the possible assignments from the part of the structure that has so far been accessed, and only one object that is distinct from those previously encountered during the last execution of `repOk`. In fact, trying more than one such object amounts to making equivalent assignments since they result in isomorphic structures [4]. Indeed, there is little reason to explore more than one structure from a set of isomorphic structures since they are either all valid or all invalid[3].

Our repair algorithm builds on our previous work on test input generation using Korat [4] and generalized symbolic execution [18], and adapts those algorithms to perform efficient data structure repair.

5 Implementation

Our prototype, Juzi, is written in Java and works for repairing structures that a Java program manipulates. There are three key inputs to Juzi: (1) the name of the class to which the structure belongs; (2) the name of the method that represents the desired structural constraints; and (3) the name of a method which represents the corrupted structure. To systematically modify field values and to perform symbolic execution, Juzi performs instrumentation of Java bytecode and implements a simple backtracking algorithm. Juzi uses CVC Lite [2] to determine feasibility of path conditions that it builds during symbolic execution.

5.1 Bytecode Instrumentation

There are two basic functions that Juzi performs using bytecode instrumentation: (1) systematic assignment of values to fields; (2) symbolic execution.

Recall that our repair algorithm uses a systematic assignment of values to fields, where some values may be symbolic. How these assignments are made depends crucially on the order in which fields are accessed. To record this order, Juzi transforms the original code and replaces field accesses by invocations of methods that Juzi adds to the given code[4]. The method invocations allow Juzi to record field accesses. For example, for the doubly-linked list example (Section 2.2), the Java bytecode statement

```
6:   getfield      #18; //Field header:Ljuzi/examples/LinkedList$Entry;
```

which accesses the `header` field, transforms to

```
6:  invokevirtual #252; //Method _get_header:()Ljuzi/examples/LinkedList$Entry;
```

which invokes the method `_get_header`—a method that Juzi adds to allow monitoring `repOk`'s executions.

To enable symbolic execution, Juzi replaces primitive types by library types that it provides to represent expressions on symbolic (and concrete) values. Juzi performs a conservative reachability analysis to see what types need to be transformed and generates appropriate bytecodes. Juzi also transforms operations on primitive values into

[3] We assume that `repOk` uses actual object identities only in comparison operations [4, 22].

[4] We used a similar approach in previous work that used source-code instrumentation to perform test generation [4, 18, 22].

appropriate method invocations. For example, when a primitive integer constant forms a sub-expression in an expression on symbols and concrete values, Juzi wraps the integer constant in an object. As an illustration, consider the following sequence of Java bytecodes:

```
...
16:  iconst_3
17:  iadd
...
```

is transformed to

```
...
16:  new     #280; //class IntConstant
19:  dup
20:  iconst_3
21:  invokespecial  #283; //Method juzi/expr/literal/IntConstant."<init>":(I)V
24:  invokevirtual  #296; //Method juzi/expr/Expression.iadd:
                          //    (Ljuzi/expr/Expression;)Ljuzi/expr/Expression;
...
```

which shows the wrapping of the integer constant 3 into an object of the library class `juzi.expr.literal.IntConstant`, followed by an invocation of the library method `iadd`, which is one of the methods Juzi implements to build expressions containing symbols and concrete values.

To allow symbolic execution to explore different program paths, Juzi uses a non-deterministic boolean choice whenever there's a branch in bytecode that cannot be deterministically evaluated on-the-fly.

Juzi uses the Java Programming Assistant (Javaassist) [6] to perform bytecode instrumentation.

5.2 Backtracking

Juzi implements a simple backtracking algorithm to provide non-deterministic choice. The class `Explorer` provides method `choose` which takes an integer input and represents a non-deterministic choice, for example the assignment

```
x = Explorer.choose(3);
```

non-deterministically assigns the values $0, 1, 2, 3$ to `x`. Such non-deterministic choice operators are an essential feature of software model checkers [14, 12, 29].

The Juzi backtracking algorithm performs stateless depth-first search (i.e., stores no states but remembers the values it uses when making non-deterministic assignments with `choose`). Non-deterministic code is thus re-executed from the beginning, and during each execution one of the non-deterministic assignments is made differently from that in the previous execution.

5.3 Satisfiability of Path Conditions

Juzi checks satisfiability of path conditions using the CVC Lite [2] automated theorem prover. CVC Lite provides a C++ API for checking validity of formulas over several

interpreted theories including linear arithmetic on integers and reals, arrays and uninterpreted functions. Since CVC Lite is implemented in C++, it can be expensive to make calls to it from a Java program. Juzi, therefore, implements some on-the-fly simplifications of path conditions as it builds them. The simplifications not only allow Juzi to generate smaller path conditions but also, in some cases, let it decide satisfiability without having to call CVC Lite routines. Juzi's simplifications include transforming constraints in a path condition to a canonical form, performing subsumption checking for simple cases, and propagating constants.

6 Case Study: Red-Black Trees

To illustrate the variety of constraints that our repair algorithm can handle, we next present a case study on repairing red-black trees [8], which implement balanced binary search trees. Red-black trees are structurally among the most complex of the commonly used data structures and therefore present a challenging study for repair. The experiments show that Juzi can efficiently repair red-black trees of moderately large sizes, e.g., repairing a tree with over a hundred nodes—almost all of which had at least one field with a corrupted value[5]—in less than a second. It is worth pointing out that prior work [9] on repair has not addressed repair of structures as complex as red-black trees.

All experiments reported in this paper were performed on a 1.6 GHz Pentium M processor with 1 GB of RAM.

The following code declares a red-black tree in a fashion similar to the implementation in `java.util.TreeMap`:

```
class TreeMap {
    Entry root;
    int size;

    static class Entry {
        int key;
        Entry left;
        Entry right;;
        Entry parent;
        boolean color;
    }

    boolean repOk() { ... }
}
```

A tree has a `root` entry and stores the number of entries in the `size` field. An entry stores a data element in the field `key`, has a `left` and a `right` child, and also has a `parent` pointer. Furthermore, an entry has a color, which is either `RED` (false) or `BLACK` (true).

Red-black trees are binary search trees. In addition to acyclicity and correct search order on keys, there are two fundamental constraints that define red-black trees:

- red entries have black children;
- the number of black entries on any path from the root to a leaf is the same.

[5] Less corrupt structures are repaired even more efficiently.

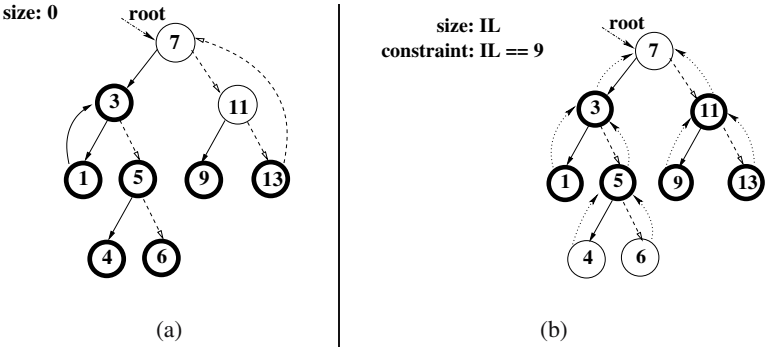


Fig. 3. Repairing a red-black tree. (Solid arrows represent `left` fields; dashed arrows represent `right` fields; dotted arrows represent `parent` pointers; `root` field is labeled; `key` is written inside the entry; entries drawn with thick circles are black and the others are red.) (a) A corrupted red-black tree structure: has cycles; has variable number of black entries along different paths from root; has a red entry with a red child; has incorrect size. (b) Tree resulting after repair has been performed: field values have been modified to satisfy the structural constraints; the `size` field is correctly constrained to be exactly 9.

Of course, the value of the `size` field needs to correctly reflect the number of entries.

Consider the corrupted structure shown in Figure 3 (a). Not only is it not acyclic, but it also violates both the constraints on the coloring of entries, has an incorrect value for the `size` field, and has all the `parent` pointers set incorrectly to `null`.

Given this structure and `repOk` for `TreeMap` (which we do not present here due to brevity), Juzi produces the structure shown in Figure 3 (b). Notice that all fields now have correct values; the value of `size` field is correctly constrained to equal 9. Juzi completed the repair in a tenth of a second.

7 Discussion

We next discuss some characteristics and limitations of our approach and present some promising future directions.

7.1 Structure Repair Versus Structure Generation

We view structure repair as being closely related to structure generation and therefore closely related to test input generation from input constraints [17, 22]. Generation addresses the problem of generating structures that satisfy given structural constraints, while repair addresses the problem of generating a structure that not only satisfies the constraints but is also heuristically close to a given corrupt structure. Interestingly, on the one hand, generation can aid repair since any structure that satisfies the constraints is indeed a candidate for being the repaired structure, while on the other hand, repair can aid generation since an arbitrarily selected corrupt structure may be repaired to generate a desired structure.

7.2 On-demand Symbolic Execution

Our use of symbolic execution is non-conventional not only in the sense of our application to data structure repair, but also in the sense of how we perform it. Symbolic execution is usually performed either by treating all program inputs as symbolic [19] or by a priori determining which inputs to treat as symbolic and which to treat as concrete (e.g., symbolic primitives and concrete references [18]).

Our repair algorithm takes a different approach. It starts by executing `repOk` (class invariant) on a structure, all of whose fields have concrete values. During subsequent executions of `repOk`, the algorithm makes values of certain fields symbolic. However, the values of these fields do not have to stay symbolic during all subsequent executions—a field may regain a concrete value (since backtracking re-initializes field values). The algorithm, thus, performs symbolic execution on an as-needed basis and whether it treats a field as symbolic or concrete depends on the particular execution being considered. This hybrid approach enables the algorithm to explore structures in a "neighborhood" of the original structure and generate a new structure that is heuristically similar to the original one.

7.3 Sensitivity of Repair to `repOk`

Repair actions performed by our algorithm intrinsically depend on how `repOk` is formulated. Recall that the algorithm backtracks on the *last* field accessed by `repOk` and modifies this field. This means that for the same corrupted structure, two different `repOk` implementations that access fields in different orders may cause our repair algorithm to produce different structures. Even though this sensitivity to the way constraints are written may be considered an inherent limitation of the algorithm, in fact, it allows the user to control how the structure may be repaired. By ordering constraints appropriately the user can ensure that the algorithm will not perturb the values of certain fields (that the user's deems unlikely to be corrupted) unless absolutely necessary.

7.4 Repairing Primitive Data Values in a Structure

The question of how to repair primitive data values in a structure is rather important for any repair algorithm. For example, consider repairing a binary search tree whose elements are not in the correct search order. One way to repair this structure is to replace the elements with new elements that appear in the correct search order. However, this is unlikely to be a good repair choice, e.g., consider the case when the tree is implementing a set—it is the elements that define the set and are therefore of crucial significance.

Our approach is to allow the user to specify ranges of data values for primitive fields and to use these ranges to constrain the repaired values of these fields. Juzi reads these ranges from a configuration file. The user can choose not to provide any range, in which case Juzi (by default) tries to preserve as many of the original values as possible. We plan to allow the user to state specific relations between (values of) a corrupted structure and a repaired structure akin to specifying post-conditions that relate pre-state with post-state. A more sophisticated approach could define metrics of similarity between corrupted and repaired structures; these metrics could then be used as a basis of new algorithms that produce repaired structures that are maximally similar to the given corrupted structures.

7.5 Converting Symbolic Values to Concrete Values

Our repair algorithm uses a decision procedure for evaluating feasibility of path conditions. A caveat of using an automated theorem prover for this purpose is that such tools typically report only the feasibility of constraints and not actual valuations of the variables in feasible constraints. This implies that when we repair the value of a primitive data field, we need to perform an additional step of selecting a concrete value. In the benchmarks we have shown, selection of such values has just been a trivial step. However, in other cases when constraints are more complex, it is non-trivial to select these values.

7.6 Multi-threaded Programs

Corrupted data in multi-threaded programs can be repaired by suspending processes that manipulate this data, repairing the data using the repair routine, and resuming the processes. This requires, however, control over thread scheduling, which cannot easily be achieved for arbitrary programs running under a standard virtual machine. Programs where it is crucial to maintain essential structural integrity constraints can, however, be run under environments that provide such suspend/resume mechanisms.

7.7 Structure Repair and Program Debugging

Repair of a structure can lend useful information about error localization in a faulty program to aid its debugging. For example, if repair actions only set values of the fields `previous` and `size` (as in Section 2.2), the user can start debugging by first looking at those parts of the code that modify these fields.

7.8 Repairing Large Structures

Our repair examples so far have involved small structures. Repair can, however, be performed feasibly for modestly large structures. For example, for doubly-linked list and binary tree (Section 2), Juzi can repair structures with over 1000 nodes—of which over a 100 have some field that needs repair—in under a minute. Structures with over 100 nodes—of which over 50 have some field that needs repair—are repaired in about a second, even in the case of red-black trees (Section 6). These results are encouraging as they point out that repair routines can efficiently be included in code where it is essential to enforce structural integrity constraints at key control points. Whether a generic repair approach can scale to repairing structures with millions of nodes—of which thousands have some field that needs repair—well, that remains to be seen.

8 Conclusions

We have presented a novel algorithm for repairing structurally complex data. Given an assertion that represents desired structural integrity constraints and a structure that violates them, the algorithm performs repair actions that mutate the given structure to generate a new structure that satisfies the constraints. Assertions are written as imperative predicates that can express rich structural properties. Since these properties can be arbitrarily complex, our algorithm is sound but not complete.

Experimental results with our prototype, Juzi, show that it is feasible to efficiently repair a variety of complex data structures that are used routinely in library code. Juzi can often repair structures with over a hundred objects (where majority of the objects have at least one field that has been corrupted) in less than one second.

Our algorithm is based on systematic backtracking but does not require storing states and can easily be implemented in a variety of software model checkers, such as the Java PathFinder, SPIN, and VeriSoft.

Acknowledgments

We would like to thank the anonymous reviewers and Darko Marinov for detailed comments on a previous draft. This work was funded in part by NSF ITR-SoD award #0438967 and in part by the GEM fellowship.

References

1. William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *Proc. 17th ACM Symposium on Operating Systems Principles (SOSP)*, Kiawah Island, December 1999.
2. Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proceedings of the 16th International Conference On Computer Aided Verification*, Boston, MA, July 2004.
3. Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Masahiro Fujita, and Yunshan Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proc. 36thConference on Design Automation (DAC)*, New Orleans, LA, 1999.
4. Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, pages 123–133, July 2002.
5. Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, June 2002.
6. Shigeru Chiba. Javassist—a reflection-based programming wizard for Java. In *Proceedings of the ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, October 1998.
7. James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. 22nd International Conference on Software Engineering (ICSE)*, June 2000.
8. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.
9. Brian Demsky and Martin Rinard. Automatic detection and repair of errors in data structures. In *Proc. ACM SIGPLAN 2003 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 78–95, 2003.
10. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proc. ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, 2002.
11. Iván García. Enabling symbolic execution of Java programs using bytecode instrumentation. Master's thesis, Department of Electrical and Computer Engineering, The University of Texas at Austin, May 2005.

12. Patrice Godefroid. Model checking for programming languages using VeriSoft. In *Proc. 24th Annual ACM Symposium on the Principles of Programming Languages (POPL)*, pages 174–186, Paris, France, January 1997.
13. G. Haugk, F. Lax, R. Royer, and J. Williams. The 5ESS(TM) switching system: Maintenance capabilities. *AT&T Technical Journal*, 64(6 part 2):1385–1416, 1985.
14. Gerald Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), May 1997.
15. Daniel Jackson. Micromodels of software: Modelling and analysis with Alloy, 2001. `http://sdg.lcs.mit.edu/alloy/book.pdf`.
16. Daniel Jackson and Mandana Vaziri. Finding bugs with a constraint solver. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, Portland, OR, August 2000.
17. Sarfraz Khurshid. *Generating Structurally Complex Tests from Declarative Constraints*. PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, December 2003.
18. Sarfraz Khurshid, Corina Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Proc. 9th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Warsaw, Poland, April 2003.
19. James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
20. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report TR 98-06i, Department of Computer Science, Iowa State University, June 1998.
21. Barbara Liskov and John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.
22. Darko Marinov. *Automatic Testing of Software with Structurally Complex Inputs*. PhD thesis, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 2004.
23. Darko Marinov and Sarfraz Khurshid. TestEra: A novel framework for automated testing of Java programs. In *Proc. 16th IEEE International Conference on Automated Software Engineering (ASE)*, San Diego, CA, November 2001.
24. Samiha Mourad and Dorothy Andrews. On the reliability of the IBM MVS/XA operating system. *IEEE Transactions on Software Engineering*, 13(10):1135–1139, 1987.
25. Martin Rinard. Resilient computing. Technical report, MIT Computer Science and Artificial Intelligence Laboratory, 2003. (Research Abstract).
26. Alexey Smirnov and Tzi cker Chiueh. DIRA: Automatic detection, identification, and repair of control-hijacking attacks. In *The 12th Annual Network and Distributed System Security Symposium*, San Diego, CA, February 2005.
27. Yuk Lai Suen. Automatically repairing structurally complex data. Master's thesis, Department of Electrical and Computer Engineering, The University of Texas at Austin, May 2005.
28. United States Nuclear Regulatory Commission. *Fault Tree Handbook*, 1981. NUREG-0492.
29. Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model checking programs. In *Proc. 15th IEEE International Conference on Automated Software Engineering (ASE)*, Grenoble, France, 2000.

A Structural Invariants for Binary Tree

The following code gives the `repOk` method for `BinaryTree` (Section 2.1) [4, 22]:

```
boolean repOk() {
    if (root == null) // check that empty tree has size zero
        return size == 0;
    Set visited = new HashSet();
    visited.add(root);
    java.util.LinkedList workList = new java.util.LinkedList();
    workList.add(root);
    while (!workList.isEmpty()) {
        Node current = (Node)workList.removeFirst();
        if (current.left != null) {
            // checks that tree has no cycle
            if (!visited.add(current.left))
                return false;
            workList.add(current.left);
        }
        if (current.right != null) {
            // checks that tree has no cycle
            if (!visited.add(current.right))
                return false;
            workList.add(current.right);
        }
    }
    if (visited.size != size) // check that size is consistent
        return false;
    return true;
}
```

Crafting a Promela Front-End with Abstract Data Types to Mitigate the Sensitivity of (Compositional) Analysis to Implementation Choices

Yung-Pin Cheng

Department of Information and Computer Education,
National Taiwan Normal University,
Taipei 106, Taiwan
ypc@ice.ntnu.edu.tw

Abstract. Recently, an active research topic in software verification is applying model checkers to programs, such as multi-threaded Java code. However, a program typically consists of more behaviors, such as operations on complicated data structures or implementation details which are typically made for some criteria like performance. A brute-force model extraction may produce a poor model for analysis engine. In this paper, we give examples to show how subtle changes in implementation may result in considerable differences in analysis, particularly to compositional analysis. Unfortunately, these implementation choices are made by programmers – people who typically do not possess the knowledge of verification. To mitigate such sensitivity, we advocate that verification tools should recognize and support abstract data types and, in the meantime, prohibit or suppress the use of array. Programming process behaviors with abstract data types can hide and converge the implementation choices. More importantly, abstract data types are informative. They provide essential information for tool automation to select a best implementation for analysis. In this paper, we describe the design and implementation of such a prototype tool which can parse systems written in Promela syntax.

1 Introduction

Automatic verification techniques such as model checking have been viewed as a promising method to ensure the quality of complicated systems. Many hard-to-detect errors, such as deadlocks, can be manifested by these techniques. In past decades, considerable progress has been made in these techniques. Several prototype tools, such as SPIN[14][15], SMV[19], have been built and applied to many software or hardware systems. In this paper, we focus on software verification.

Software typically has more states than hardware. The wide variety of software designs make software verification a more difficult task than hardware. For example, famous Ordered Binary Decision Diagram (OBDD) [3] which is widely used in hardware verification has no obvious merits in software verification (see Corbett's work in [10]). Besides, modeling a software system requires much more efforts, experiences, and human wisdom. In that work [10], Corbett also found subtle modeling differences for the same system may produce obvious differences in the results of analysis in different tools. However, the objective of Corbett's work is to compare performance of

P. Godefroid (Ed.): SPIN 2005, LNCS 3639, pp. 139–153, 2005.

various verification tools, so, finding and eliminating these differences to have a fair comparison of verification tools are his first priority.

Generally, most verification tools share the similar technology in exploring reachable states. They can be used for either software verification or hardware verification. Nevertheless, software verification researchers often prefer one tool over another for its capability of modeling software. For example, Spin [14] provides a model description language (MDL) called Promela, which has syntax close to a high-level programming language. Although it is originally designed for modeling communication protocols, many researchers have chosen it to verify concurrent programs written in Ada, Java and C. However, verifying systems written in these programming languages using Spin can be subtle. The abstraction of programs into Promela models requires human wisdom and experiences and is error-prone. In [16], Holzmann argued that a blindly derived model (for example, either by a naive automatic/manual extraction) is unlikely to work for verification in most cases. In other words, constructing an efficient and correct model requires human wisdom from experienced personals.

Several years ago, some works rose to the challenge. Research tools such as Bandera [11] and Pathfinder[13] have been developed to automatically extract models from Java source code. Their goal is to model-check Java source code by smartly extracting a model from Java code for analysis engines like SPIN. Bandera also introduces slicing techniques to abstract away the program behaviors that do not concern the interested properties (particularly liveness properties) so that the state explosion problem can be alleviated. In our opinions, these progresses mark an important milestone for automatic software verification.

Despite the progress described above, the fundamental barriers of software verification, however, still remain. Verification tools which analyze all processes at once are inevitably limited by the PSPACE lower bound in worst case; that is, the number of reachable states grows exponentially as the number of processes increases. In other words, any attempt to alleviate the state explosion is bound to fail in general but may work for some cases. For example, Bandera uses property to guide the program slicer to slice away the behaviors that are not concerned by the property, particular the liveness properties defined in linear-time temporal logic (LTL) formula. Such approach does not work for property like freedom of deadlocks. Reachable deadlocks must be manifested from all the possible behaviors.

To tackle the state explosion problem, a more promising approach is compositional analysis[7,6,8,9,12]. Compositional analysis avoids state explosion by dividing a whole system into many subsystems. Then, the techniques described above are used to analyze these subsystems. Ideally, the analysis of each subsystem would produce manageable and smaller state space and then each subsystem can be replaced by a simple interface process. The process is continued by combining the analysis of subsystems into a larger subsystems in a hierarchical fashion until the whole system is analyzed. Unfortunately, this ideal scenario seldom happens in practical cases. Compositional analysis is architecture sensitive. In many systems, no feasible hierarchies exist in their as-built architecture; that is, The power of divide-and-conquer is often limited by the system architecture.

In this paper, we describe the design and implementation of a Promela front-end. This front-end is part of a compositional analysis tool suite which is under developing. The major feature of this front-end includes the new statements for refactoring[1] a process behaviors to overcome the problem of architecture sensitivity of compositional analysis. Another new feature of this front-end is the support of abstract data types. From our past experiences, we discovered a program (either written in Java, C, or Ada) may be written in a way that is poor[2] for analysis engine, particularly when abstract data types are implemented by array. We show two functionally equivalent process behaviors with two implementation choices can produce significant differences in analysis, particularly to compositional analysis with refactoring. To address the problem, we propose an extension of Promela. In this extension, we add abstract data types such as *queue* and *set* to its syntax. We show that encouraging the use of abstract data types and prohibiting the use of array can limit the wild implementation choices a programmer may make, therefore, mitigating the sensitivity of analysis. Furthermore, abstract data types are informative, providing essential information for tools to determine the best implementation for analysis without the need of code analysis.

Note that Spin is a sophisticated piece of software. Our objective is not to rework its features. Our long term goal is the construction of a software verification tool suite which is compositional-oriented. We select Promela as one of our input language because of its syntax simplicity and its popularity. This paper is organized as follows. In section 2, we give an overview of compositional analysis and our refactoring technique. In section 3, we give examples to explain why analysis is sensitive to implementation choices. Section 4 describes our design and implementation of a prototype tool. Finally, we end the paper with discussion, related work, and conclusions in section 5 and 6.

2 An Overview of Compositonal Analysis and Refactoring

In this section, we give an overview of two techniques, compositional analysis and model refactoring, so that readers can have a brief idea on the problem we want to address in this paper.

2.1 Compositional Analysis

In a compositional analysis, we often have to group a set of processes into a subsystem (or a module). There are two basic criteria of a "good" subsystem. First, the processes inside the subsystem must not generate excessive state space. Second, the subsystem's state space must be able to be replaced by a much simpler *interface process* to represent the subsystem's state space. An interface process can be computed automatically by hiding internal interactions, minimizing the state space, and exporting the state and transitions (a.k.a interfaces) that will be used by its environment. Note that exporting state and transitions as interfaces can aggregate the state explosion problem if the interfaces are not simple (see [12]). So, simple interface is the key to a "good" subsystem.

[1] This technique will be explained later.

[2] Note that a program may be written in a way that is poor for analysis but is good for performance or other measuring criteria.

In other words, an effective subsystem should be *loosely coupled* to its environment so that the chance of having a simple interface process to replace it in compositional analysis is higher. At last, "good" subsystems and processes must produce another larger "good" subsystem in the composition hierarchy until the whole system is analyzed. Unfortunately, this ideal scenario seldom occurs in the compositional analysis of large and complicated systems.

2.2 Model Refactoring

In Fig. 1(a) and Fig. 1(b), we show the state graphs of three example processes *X,Y,* and *S* in CCS semantics [20] (where synchronization actions are matched in pairs) and their synchronization structure. Such kind of structure, a star-shape structure, appears very often in practice, for example, a stateful server which communicates with clients via separate (or private) channels. Many systems can even have structures of multiple stars.

We say *S* is *tightly coupled* to its environment (which consists of *X* and *Y*) because it has complicated interfaces to its environment. Suppose *S* is a server and *X,Y* are clients. Image the number of clients is increased to a larger number. Any attempt to include *S* as a subsystem is bound to fail because of the complicated interfaces to its environment. That is, no feasible subsystems and composing hierarchies exist in this structure, particularly when client number is large.

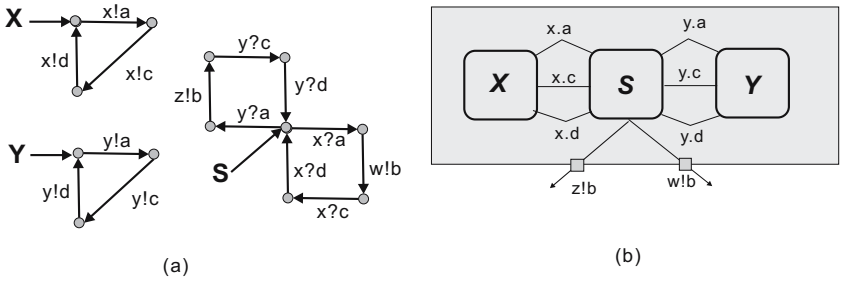


Fig. 1. (a) A simple example with 3 processes X, Y, and S. (b) The synchronization structure of the example.

In [4,5], we proposed an approach called *model refactoring* to enable compositional analysis for systems which are originally prohibited by their as-built architecture. The refactoring consists a set of transformations. Each transformation maintains the behavioral equivalence (weak bisimulation) of the model. By applying a sequence of transformations, a model P is gradually transformed into a model P' with new structure which is more amenable to compositional analysis. It consists in building a sequence of equivalent models, each obtained by the preceding ones by means of the application of a rule. The rules are aimed for restructuring the as-built structures which are not suitable for compositional techniques. The goal is to obtain a transformed model whose structure contains loosely coupled components, where processes in each component do not yield state explosion.

The key transformations are to decompose centralized, complicated behaviors of a process into several small new processes while behavioral equivalence is preserved. In [5], we described the basic tool support[3] for refactoring and showed that a refactored elevator system can be analyzed up to hundreds of elevators but global analysis and compositional analysis (without refactoring) can only analyze up to 4 elevators.

For instance, we show the refactored *X,Y,* and *S* in Fig. 2(a) and the new synchronization structure in Fig. 2(b). In Fig. 2(a), the behaviors related to channel *x* (or to process *X*) is removed and wrapped into a new process *Sx.* Similarly, the behaviors related to channel *y* is removed and wrapped into a new process *Sy*. So, the rendezvous of *x!a, x!c,* and *x!d* are now redirected to *Sx*. However, *Sx* and *Sy* are now two individual processes which can execute concurrently, but their original joint behaviors in *S* can not. So, extra synchronizations (*e!lock and e!release*) are inserted to maintain behavioral equivalence; that is, before invoking *x!a* and *y!a, X* and *Y* are forced to invoke *e!lock* first. Then, at the end of *Sx* and *Sy*, *e!release* is used to free *S*.

The idea of refactoring equivalence is easy to explain. Let's image the modified processes (*X,Y*, and *S*) are contained in a black box. Image you are an external observer of the black box. The external behaviors of the black box are defined by *z!b* and *w!b*. In Fig. 1(b), the black box (which we call it B1) is implemented by 3 processes. The black box (we call it B2) in Fig. 2(b), on the other hand, is implemented by 5 processes. The external behaviors are also defined by *x!b* and *y!b*. Our refactoring must ensure the external behaviors are equivalent before and after a transformation. Intuitively, B1's external behaviors can be viewed as an specification. Then, we choose to implement the specification with 5 processes. Since we use 5 processes to do the same work which was originally done by 3 processes, extra communications for process coordination are inevitable. As long as the extra synchronizations are restricted inside the black box, the two black boxes behave equivalently to an external observer.

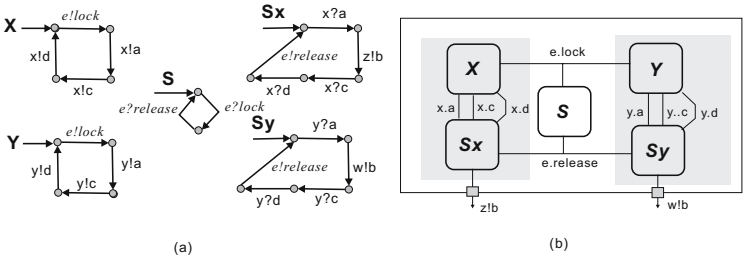


Fig. 2. The refactored example system

2.3 Tool Support

From the above example, it may looks like identifying the behaviors and decomposing them can be done at the finite-state representation. In practice, automatic refactoring does not work at this level of representation. Some important information needed by

[3] The tool support can successfully refactor many systems in an automated fashion, particularly the behavioral patterns which do not involve complicated data structures.

refactoring engine is lost at this level. The refactoring automation must be made when a CCS state graph is created. So, a refactoring statement is added to Promela's syntax. We chose a subset of Promela syntax and built a parser to translate Promela code into CCS [20] state graph. For example, *S* in Fig. 1 can be written in Promela as follows:

```
mtype = { a,b,c,d } ;
chan x = [0] of mtype ; chan y = [0] of mtype ;
chan z = [0] of mtype ; chan w = [0] of mtype ;
proctype S() {
   do (0)
   refactorby x, y {
   :: x?a (1) ->
      z!b (2); x?c (3); x?d (4);
   :: y?a (5) ->
        w!b (6); y?c (7); y?d (8);
   }
   od
}
```

Note that in the example, we mark a statement with "*(addr)*" as the address of each statement. To translate a Promela code into a CCS state graph is trivial. First, we collect the values of local variables and the address of current statement to make a tuple like $(v_1, v_2, ..v_n, addr)$, where v_i are values of local variables and $addr$ is the current statement address. In process *S*, there are no local variables, so the only element in the tuple is statement address *(addr)*. Since the process starts at statement address 0, so we use (0) as initial state. We begin parsing the abstract syntax tree (AST) of the code. When we parse a channel statement which sends or receives a message, we create a new outgoing transition to a new state. The new state has its $addr$ updated to next statement address and the outgoing transition has label in the form of "*ch?msg*" or "*ch!msg*." The traversal is continued until no more new states are explored.

To activate refactoring, tool users can add keyword *refactorby* to enclose a block of statements they want to refactor. For example, to obtain the result in Fig. 2(a) and Fig. 2(b), we use *refactorby* to notify refactoring to separate the behaviors by channel name. In general, process behaviors can be distinguished by channel name, variable's values, etc.

When refactoring mode is activated, we translate Promela code into segments of behaviors. For example, the sequence of transitions beginning from *x?a* and ending with *x?d* is called a segment. These segments are grouped according to group options, the parameters behind keyword ***refactorby***. Next, segments are wrapped into a new process such as *Sx* in Fig. 2(a) by a unified transformation.

Note that, in principle, it is impossible for our transformation to decompose any process behaviors and make compositional analysis work in general, otherwise, we would have solved the notorious state explosion problem. So, it is easy for a malicious tool user to write a peculiar process behaviors which makes refactoring fail. However, under normal circumstances, most process behaviors are written in common patterns. Our ultimate goal is to make refactoring work for most behavioral patterns.

3 Sensitivity of (Compositional) Analysis

In the past, we have successfully refactored several systems. Most of them appear as examples in literatures, such as elevator system[21], furnace system[22], alternating bit protocol[2], etc. The tool support described in the previous section is sufficient for many systems whose process behaviors either have no presence of data values or only have simple data values to enrich its behavioral patterns. On the other hand, we began to encounter systems with behaviors complicated by array. When process behaviors are complicated by array, segmented behaviors may be interwined and tangled and refactoring transformations are no longer feasible. We described some of the behaviors as follows.

3.1 Chiron User Interface System

The first example is called Chiron user interface [17]. It has been analyzed by [1,23]. Chiron user interface system is originally written in Ada. Chiron's design philosophy is to separate application code from user interface code. So, there are user interface agents called *artists* attached to selected data[4] belonging to the applications. At runtime, each artist can register *events* of interests to *dispatcher.* Whenever there is an operation call on the data, the dispatcher intercepts the call and notifies each of the artists associated with that data with the event.

Its Promela model is manually extracted from its Ada source code. The most complicated process in Chiron is a task called *dispatcher*. *Dispatcher* is responsible for accepting requests to register or unregister an event from an artist. The dispatcher use an array *e1_list*

```
mtype e1_list[no_of_artists];
```

to keep track the artists which have registered on event *e1*. When an artist registers an event *e1* to dispatcher, the following code fragment is executed in dispatcher.

```
dispatcher_chan? register_event, artist_id, event ->
if
:: (event == e1) ->
     i = 1 ;
     do
     :: if
        :: (i> e1_size) ->
           e1_size ++ ;
           e1_list[i-1] = artist_id ;
           break ;
        :: else
        fi;
        if
        :: (e1_list[i-1] == artist_id) ->
           break ;
        :: else
        fi;
        i++ ;
     od
```

[4] You can consider the data as an object and the object's values (or attributes) is linked to a visualization tool called artists. In other words, an artist can be viewed as a graphic drawing unit for the data.

The code first receives a command and two parameters from the channel. Two parameters are *artist_id* and *event*. Next, it checks if the *artist_id* is already in the array, using a loop index *i*. If not, the *artist_id* is appended to the tail of the array.

On the other hand, to unregister event *e1* from dispatcher by an artist, the following code is executed.

```
dispatcher_chan? unregister_event, artist_id, event ->
if
:: (event == e1) ->
   if
   :: (e1_size == 0) -> skip
   :: else ->
      i = 1 ;
      do
      :: (i> e1_size) -> break ;
      :: else ->
         if
         :: (e1_list[i-1] == artist_id ) ->
            do
            :: (i>= e1_size) -> break ;
            :: else ->
               e1_list[i-1] = e1_list[i] ;
               i++ ;
            od
            e1_size -- ;
         :: else
         fi ;
         i++ ;
      od;
      e1_size[e1_size] = 0 ;
   fi;
fi;
```

The code first search the array to check if the *artist_id* is in the array. If yes, the element (pointed by *i*) is deleted and all the elements behind *e1_list[i]* is copied to fill the deleted space. In other words, the elements in *e1_list* are shifted. To anyone who know programming, such implementation is only one of many choices. Typically, if we prefer such kind of implementation, we want to maintain the order of artists by their registration time. That is, an artist which registers *e1* earlier is stored in the front of array. However, in dispatcher task, we found no clues where such order is concerned.

3.2 Implementation Alternative

Since the order of registration is not a concern to dispatcher, a better implementation choice is using a bit array.

```
bit e1_list[no_of_artists];
```

In this implementation, if *e1_list[i]* = 0, it means artist a_i does not register on event *e1*. If *e1_list[i] = 1*, it means artist $a_i$has registered on event *e1*.

3.3 Analysis of Implementation Choices

In programming, we are accustomed to make implementation choices for some reasons, perhaps for performance or maintenance. Similarly, the above two implementation choices produce two functionally equivalent models but unfortunately, result in

great difference in analysis. Let the length of array *e1_list* be n, the number of artists. We call the array of original dispatcher as queue array. The original dispatcher's behaviors can produce states which have growing rate proportional to

$$1 + \sum_{i=1}^{n} \binom{n}{i} i!.$$

On the other hand, using bit array has a growing rate proportional to 2^n. Although the two scales are both exponential, the first growing rate is much worse than the second one for global analysis.

To compositional analysis, the implementation with queue array produce interwined and tangled behaviors which cannot be refactored effectively. It can be only analyzed up to 2 artists. On the other hand, the behaviors with bit array can be refactored effectively into loosely coupled components. Its refactored structure can fully take the advantage of divide-and-conquer. It can be analyzed up to 14 artists. Note that, in Chiron, increasing an artist means adding a new process to the system.

In Fig.3(a), we show the tangled behaviors of the queue array implementation with two artists. In the figure, a registration event is abbreviated into "?Rx" where x is the type of event. An unregistration event is abbreviated into "?Ux." Beside each state, we print the contents of array *e1_lst[]*. On the other hand, Fig. 3(b) shows the behaviors using bit array, which presents some form of symmetry. This behavior can be effectively transformed by refactoring (It uses value processes to model value change for each array element *e1_lst[i]*. Readers who are interested in these technical details, please refer to [5]).

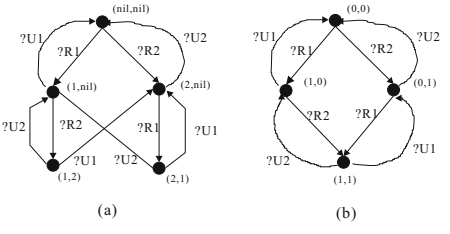


Fig. 3. The behavioral patterns of two implementations

This observation agrees with Holzmann's statement [16] that naively translated or blindly derived models are unlikely to work for analysis in most cases. The key points of our observation are:

- **Programmers often make implementation choices for performance or other criteria, but not for analysis. A model which is directly extracted from a program inherits the program's implementation choices, which can be "poor" for analysis but makes no significant difference in runtime execution.**
- **Subtle changes in implementation may produce significant differences in (compositional) analysis. Analysis tends to magnify small and slight implementation changes.**

So, with these observations and the state explosion problem, we believe model checking programs is just a beginning. We should be cautious and conservative on the the general applicability and practicability of these tools.

3.4 Gas Station

To convince that Chiron's case is not unique, we give another example from a gas station system. However, due to the limits of space, the content of this subsection is shown in a long version of this paper which can be downloaded from URL link *http://www.ice.ntnu.edu.tw/~ypc/ArChat.htm*

4 Using Abstract Data Types to Mitigate Sensitivity of Analysis

The problems described in the previous section can be the tip of the iceberg. Analysis tools have always been geared towards being adopted by industry to assure high software quality. However, if an analysis tool must depend on the virtue of the code or limit itself to trained experts, the fruit of software verification research will always be limited in research community.

To address the problem, the first approach we tried is attempting to analyze the array usage in the code and gather useful informations for refactoring automation. Suppose we can analyze and understand the semantics of Chiron's dispatcher task mechanically, we can replace it with bit array to produce best analysis results. Unfortunately, that is hard and impractical. We re-analyze the essence of the problem and have three observations, which are:

1. Array is one of the very basic blocks for constructing many abstract data types (ADTs). Most process behaviors with array operations can be summed up to some kinds of ADT operations.
2. An ADT may have several implementation choices. However, these implementation choices can be hidden by ADT interfaces.
3. The process of using an ADT for a task encourages precise and high-level thinking.

These observations are the basis of this work. Image an extreme scenario where analysis tools are integrated into a programming environment. A programmer is responsible for a critical task which requires concurrent programming. Under this condition, array is prohibited by the environment because the code must be analyzable. A programmer would be forced to select appropriate ADTs to complete his work. In the case of Chiron's dispatcher, he would select *set* as the most appropriate ADT for the job.

In the scenario, the usage of *set* provides explicit directives for tool automation. Selecting best implementation (i.e., bit array in this case) becomes straightforward. There is no need to incorporate other static analysis techniques for program comprehension. Consequently, sensitivity to implementation choices is controlled and mitigated. In this paper, we implement two frequently used ADTs into Promela to demonstrate our idea. They are:

QUEUE

```
DECLARATION SYNTAX:
   queue qname = [n] of {enumtype}
METHODS:
  void push(enumtype val); // to add a value val to the queue
  enumtype pop(); // return and remove the frist element of queue
  enumtype front(); // return the value of first element
```

SET

```
DECLARATION SYNTAX:
   set sname = [n] of {enumtype};
METHODS:
   insert(enumtype val);  // add val to a set
   erase(enumtype val);   // remove val from a set
   int find(enumtype val); // return the index of the value
```

Where *enumtype* is a type defined by **enum** keyword, another new function we add to Promela to extend **mtype** of Promela. A user can use

$$enum\ \ clien\_type = \{c1, c2, c3\};$$

to define an enumeration type in Promela. Both the ADTs are exclusive; that is, values in these containers can not be duplicated. The implementation of containers which allow duplicated elements can be quite different from the exclusive ones. Currently, exclusive ones can satisfy our need.

Using the new ADTs, the dispatcher can be rewritten into

```
enum artist_type = {a1,a2};
set e1_lst = [2] of {artist_type};
.....
dispatcher_chan? register_event, artist_id, event ->
if
:: (event == e1) ->  e1_lst.insert(artist_id);
:: (event == e2) ->  e2_lst.insert(artist_id);
fi
.....
```

In this example, not only the process behaviors are concise and easy to understand, but also process behaviors are forced to "converge" on this one. Our tool automatically select the best implementation choice for (compositional) analysis, which is transparent to tool users. With the prevalence of object-oriented programming languages nowadays, the constraint (to prohibit or suppress the use of array) may not be strong as it looks but the merits are manifold.

4.1 Object-Oriented Tool Design and Implementation

Crafting the experimental parser described in this paper requires a lot of work. This pilot prototype[5] has been worked towards to a new framework illustrated in Fig. 4. In the

[5] The old prototype (without ADT) described in [5] has been torn apart and restructured towards the structure in Fig. 4.

figure, boxes colored in grey are tools which have been developed or under developing. White boxes are tools which can be developed by other parties or will be developed by us in the future. Finally, boxes decorated with grey stripes are tools constructed by others. DOT is graph visualization tool from AT&T. Fc2tool [18] is a tool suite from INRIA, France, which consists of tools to enumerate and minimize CCS state graphs.

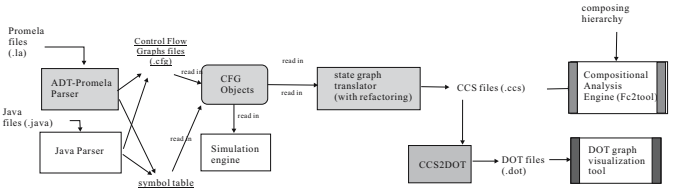


Fig. 4. The framework of tool implementation

In the framework, our ADT-promela parser reads a Promela file and produces several *cfg* (control flow state graph) files and a symbol table[6], where each process has a *cfg* file. A control flow state graph is like Fig.5(a). Each state can be interpreted as the address of a statement in the program. Each edge is then attributed by a statement. The statement is stored in the form AST (abstract syntax tree) which can be evaluated by a postfix traversal algorithm. The *cfg* file format uses tags which can be parsed and read easily. It can be modified to XML syntax if necessary.

The spirit of our design is to use files to communicate among tools. This design avoids building a monolithic tool which can be harder to modify or evolve. The framework also encourages cooperative work. Tools with a language front-end typically do not use flow graph of a program for data exchange because it is language dependent. However, we found that control flow state graph is where many tools start with. For instance, our state graph translator traverses it to generate communicating finite state machines (such as CCS or CSP state graphs). Simulation tools can use it to exercise traces. Other tools such as program slicers can work on this representation as well. These reasons make us to design it into a format which can be shared by language front-ends and analysis back-ends.

To deal with control flow state graph, we design a set of object-oriented CFG classes (shown in Fig. 5(b)), which can parse a *cfg* file to construct a control flow state graph. We introduce an inheritance hierarchy to separate what is language dependent from what is language independent. A language front-end can implement their own AST to store a statement. Next, it should implement an overidden *eval()* method (see class *CFG_edge_promela*). Other tools, such as a simulation engine, will only invoke *eval()* to evaluate the AST of a statement and update variable values in the symbol table. The details of AST (which is language dependent) are transparent to other tools. By this design, we can implement a language-independent state graph translator or a simulation engine.

[6] Spin is capable of outputting control flow state graphs and symbol tables. However, that output is not designed for the purpose like ours.

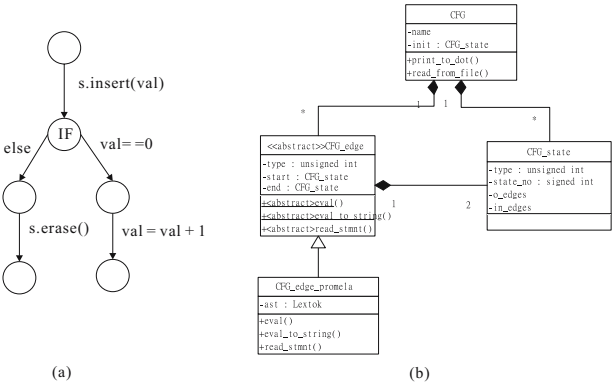


Fig. 5. (a) An example control flow state graph. (b) The inheritance structure.

The implementation of an ADT method is actually done in the overridden *eval()* of class *CFG_edge_promela*. For example, when a *set* is defined, we create a bit array in the symbol table. Later, when a statement *s.insert(i)* is evaluated, *eval()* sets the *ith* element in the bit array. In our state graph translator, bit array will be included as a tuple (see section 2.3) to traverse the control flow state graph to generate CCS state graphs.

The tools described in this paper will be gradually released at URL link *http://www.ice.ntnu.edu.tw/~ypc/ArCats.htm.*

5 Related Works and Discussion

From the best of our knowledge, Java model extractor Pathfinder has not supported abstract data types from Java standard library. In other words, it assumes the behavior of a Java thread does not involve ADTs. Bandera can process code which uses *vector*. However, *vector* is just another safe-to-use array. Supporting ADTs requires a great amount of efforts and works. The reasons they do not support ADTs from standard library are simply an issue of cost [13]. They do not address the sensitivity problem we described in this work. Besides, they focus on global analysis, whereas compositional analysis and refactoring are our major concerns.

Supporting abstract data types in Promela can be done in another way. For example, we can use CPP's macros to implement ADTs. This approach may be easier to maintain and implement. There is no need to modify Promela's grammar. However, it is more difficult to cope with object-oriented syntax nowadays. For example, many ADT objects may have methods all named *add()*. Solving naming conflicts in macro programming is more difficult. It is also more difficult to cope with our design framework in previous section.

6 Conlusions

In this paper, we describe a special phenomenon of software verification – analysis (particularly compositional analysis) is sensitive to implementation choices when array

is used to implement complicated data structures. We give examples to show that an implementation choice which is the definite choice in the view of programming may be a poor choice for analysis. On the other hand, a poor implementation choice from the view of programming can be a good choice for analysis. We show that such sensitivity can be mitigated if ADTs are supported and the usage of array are suppressed or prohibited. Two ADTs SET and QUEUE are implemented in a prototype tool. Models rewritten with ADTs have obvious advantages. First, using ADTs forces process behaviors to converge. Programmers have less room to make their own implementation choices to endanger analysis. Second, the ADTs provide useful automation information. Selecting the best implementations for ADTs behaviors becomes straightforward. There is no need to incorporate any static analysis or program comprehension techniques.

References

1. G. S. Avrunin, J. C. Corbett, M. B. Dwyer, C. S. Pasareanu, and S. F. Siegel. Comparing finite-state verification techniques for concurrent software. Technical Report UM-CS-1999-069, Dept. of CS, University of Massachusetts, November 1999. (in preparation).
2. K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A note on reliable full-duplex transmission over half-duplex lines. *Communications of ACM*, 12(5):260–261, 1969.
3. R. E. Bryant. Graph-based algorithms for boolean function manipulations. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
4. Y. Cheng. Refactoring design models for inductive verification. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA2002)*, pages 164–168, Rome, Italy, July 2002.
5. Y.-P. Cheng, M. Young, C.-L. Huang, and C.-Y. Pan. Towards scalable compositional analysis by refactoring design models. In *Proceedings of the ACM SIGSOFT 2003 Symposium on the Foundations of Software Engineering*, pages 247–256, 2003.
6. S. C. Cheung, D. Giannakopoulou, and J. Kramer. Verification of liveness properties using compositional reachability analysis. In *5th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 227–243, Zurich, Switzerland, September 1997.
7. S. C. Cheung and J. Kramer. Context constraints for compositional reachability analysis. *ACM Transactions on Software Engineering and Methodology*, 5(4):334–377, October 1996.
8. S. C. Cheung and J. Kramer. Checking safety properties using compositional reachability analysis. *ACM Transactions on Software Engineering and Methodology*, 8:49–78, January 1999.
9. E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Proceedings of 4th IEEE Symposium on Logic in Computer Sciences*, pages 353–362. IEEE Computer Society Press, 1989.
10. J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 2(3):161–180, March 1996.
11. J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.
12. S. Graf and B. Steffen. Compositional minimization of finite state systems. In *Proceedings of the 2nd International Conference of Computer-Aided Verification*, pages 186–204, 1990.
13. K. Havelund and T. Pressburger. Model checking JAVA programs using JAVA pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.
14. G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, NJ 07632, 1991.

15. G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.
16. G. J. Holzmann. Designing executable abstractions. In *Proceedings of the second workshop on formal methods in software practice*, pages 103–108, Clearwater Beach, Florida USA, March 1998.
17. R. K. Keller, M. Cameron, R. N. Taylor, and D. B. Troup. User interface development and software environments: The Chiron-1 system. In *Proceedings of the Thirteenth International Conference on Software Engineering*, pages 208–218, Austin, TX, May 1991.
18. E. Madelaine and R. de Simone. *The FC2 Reference Manual*. Available by ftp from cma.cma.fr:pub/verif as file fc2refman.ps, INRIA.
19. K. L. McMillan. *Symbolic model checking*. Kluwer Academic Publishers, Massachusetts, 1993.
20. R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1980.
21. D. J. Richardson, S. L. Aha, and T. O. O'Malley. Specification-based test oracles for reactive systems. In *Proceedings of the Fourteenth International Conference on Software Engineering*, pages 105–118, Melbourne, Australia, May 1992.
22. W. J. Yeh and M. Young. Re-designing tasking structure of Ada programs for analysis:A case study. *Software Testing, Verification, and Reliability*, 4:223–253, 1994.
23. M. Young, R. Taylor, D. Levine, K. A. Nies, and D. Brodbeck. A concurrency analysis tool suite for Ada programs: Rationale, design, and preliminary experience. *ACM Transactions on Software Engineering and Methodology*, 4(1):65–106, Jan 1995.

Behavioural Models for Hierarchical Components*

Tomás Barros[1], Ludovic Henrio[2], and Eric Madelaine[1]

[1] INRIA Sophia Antipolis, CNRS - I3S - Univ. Nice Sophia Antipolis, 2004, Route des Lucioles, BP 93, F-06902 Sophia-Antipolis Cedex - France
[2] Univ. of Wesminster, Watford Rd Northwick park, Harrow, HA1 3TP, UK
First.Last@sophia.inria.fr

Abstract. In this work, we focus on hierarchical component systems. We describe both the functional behaviour and the non-functional features (life-cycle management) of components in terms of synchronised transition systems; functional behaviours are supposed to be specified by the component developer, while management features can be built automatically for the architecture definition of a given component system. We define a notion of correct component composition; then we show how we can prove, using (compositional) model-checking techniques, temporal properties of a component system. Transformations of a system, for example replacement of a sub-component, are expressed as transformations of its behavioural semantics, allowing to prove preservation of some properties, or the validity of new properties after transformation.

1 Introduction

Components have emerged as a new programming paradigm in software development. Beyond structuring concepts inherited from modules and objects, component frameworks provide means for architecture and deployment description. Some frameworks define a number of non-functional features for controlling the life-cycle of the components and the application, or allow for construction of distributed components. In general words, a component is a self contained entity that interacts with its environment through well-defined interfaces (provided services and required functionalities to be provided by other components). Besides these interactions, a component does not reveal its internal structure.

In hierarchical component frameworks like Fractal [7], different components can be assembled together creating a new self contained component which can be itself assembled to other components in a upper level of hierarchy. Hierarchical components hide, at each level, the complexity of the sub-entities. The compositional aspect together with the separation between functional and non-functional aspects helps the implementation and maintenance of complex software systems.

* This research work is carried out under the ACI Securité FIACRE funded by the french government, and under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265)

P. Godefroid (Ed.): SPIN 2005, LNCS 3639, pp. 154–168, 2005.

The challenge that we want to address in this work is to build a formal framework to ensure that compositions are correct. Standard components systems have typed interfaces, that ensure some level of static compatibility between the components: interfaces are bound only if their operations have compatible types in the classical sense (OO method typing). This does not prevent assembled components from having non compatible behaviours, that could lead to deadlocks, live-locks, or other kinds of safety problems. A number of recent works do try to address better dynamic guaranties, e.g. research on behavioural typing or contracts [8], as well as frameworks like Wright [1] or Sofa [14].

Our approach is to give behavioural specifications of the components in the form of hierarchical synchronised transition systems. The semantics of a composite is then computed as a product of the LTSs of its sub-components with the controller of the composite. This system can be checked against requirements expressed as a set of temporal logic formulas, or again as an LTS. We aim to provide the final user with tools to verify the behaviour at the design phase (definition), the assembly phase (implementation), as well as the dynamic reconfiguration (maintenance) of the component system. Therefore the intended user of our framework is the application developer in charge of those tasks. In this work, we choose to rely on the Fractal hierarchical component model.

The models for the functional behaviour of basic components may be derived from automatic analysis of source code (involving adequate data abstraction), as we have described in [4], or expressed by the developer in a dedicated specification language, e.g. the graphical language for synchronised automata used in this paper.

Our main contributions in this paper are:

- a methodology for building behavioural models of hierarchical components, including non-structural reconfiguration operations (generated from the component ADL),
- the modelling of the full behaviour of the application as a hierarchy of parameterized LTSs,
- a specification of structural reconfigurations as transformations of the LTS expressing the component behaviour,
- a classification of correctness properties for a component system together with tools allowing and easing their verification.

Our final target is distributed component systems communicating asynchronously. We have shown in [3] how we build models for distributed objects and verify their properties; in this paper we concentrate on the modelling of the control and transformation operations of hierarchical components, and we leave for further work the integration with the asynchronous communication semantics. One example of distributed implementation of Fractal is given in [6].

In Section 2 we present the features of Fractal that will be useful for the understanding of this paper, and we introduce a small example that will serve as an illustration for the rest of the paper. Section 3 discusses the notion of correct behaviour. In section 4 we recall the main features of the formal models that we defined in [4]. Section 5 develops, step by step, the formalisation and

the behaviour computation of this example, starting with the specification of base components, then building the composite controllers, computing the composite behaviour, specifying errors, dealing with deployment and transformation phases, and finally proving in Section 6 some properties of the assembly.

2 The Fractal Component Model

The Fractal component model provides an homogeneous vision of software systems architecture with a few but well defined concepts such as component, controller, content, interface, binding. It is recursive – components structure is autosimilar at any arbitrary level (hence the name 'Fractal'); it is completely reflexive, i.e., it provides introspection and inter-cession capabilities on components structure.

2.1 Guidelines to Fractal Components

A Fractal component is formed out of two parts: a *controller* and a *content*. The content of a component is composed of (a finite number of) other components, called *sub-components*, which are under the control of the controller. This allows for hierarchic components, in the sense that components may be nested at any arbitrary level. A component that exposes its content is called a *composite* component. A component that does not expose its content, but at least one control interface, is called a *primitive* component.

The controller of a component can have *external* and *internal* interfaces. A component can interact with its environment through *operations* at its external interfaces, while internal interfaces are accessible only from the component's sub-components.

Interfaces can be of two sorts: *client* and *server*. A server interface can receive methods invocations while a client interface emits methods call. A *functional* interface provides or requires functionalities of a component, while a *control* interface is a server interface that corresponds to a "non functional aspect", such as introspection, configuration or reconfiguration.

A component controller encodes the control behaviour associated with a particular component. Fractal defines three basic (optional) levels of control capabilities for a component: no control at all, introspection, and configuration. Only the latter is of interest to us. At the configuration control level, Fractal proposes four control interfaces for each component: Attribute (to set attributes), Binding (to bind/unbind client interfaces), Content (to add/remove sub-components) and Life-Cycle (to stop/start the component).

The Fractal specification defines a number of constraints on the interplay between functional and non-functional operations. In particular: (1) content and binding control operations are only possible when and the component is stopped, and (2) when stopped, a component do not emit invocations and must accept invocations through control interfaces.

Other features are left unspecified in the Fractal definition, and may be set by a particular Fractal implementation, or left to be specified at user level. For this

paper, we make the following choices: (1) the start/stop operations are recursive, i.e. they affect the component and each one of its sub-components; (2) functional operations cannot fire control operations. (3) the controller (membrane) of composites is only a forwarder between external and internal functional interfaces without any other control capability; The last feature implies that there is exactly one internal interface for each external interface of a composite.

2.2 Component System Example

In this section we introduce a particular component system as an example, which we will use later to better explain our work. Fig. 1 is a graphical view for it.

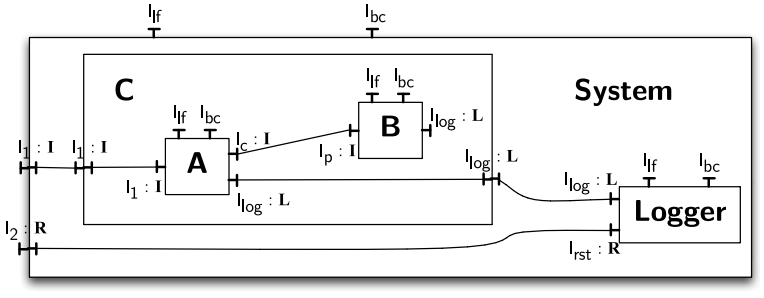


Fig. 1. A simple component system

The example is built from three *primitive* components (**A**, **B** and **Logger**), which are composed in two levels of hierarchy defined by two *composite* components (**C** and **System**). Each component exposes the interfaces for the control operations they support (in our example all the components support life-cycle control operation through the interface $\mathsf{l_{lf}}$ and binding control operations through the interface $\mathsf{l_{bc}}$).

All the functional interfaces in the example are typed either by the type **I**, the type **L**, or the type **R**. We define the type **I** having the operation foo(), the type **L** having the operation log() and the type **R** having the operation reset().

The system is deployed in a bottom-up fashion from the innermost components to the outer component (**System** in our example). At each level of hierarchy a specific deployment is applied. For instance, at the **C** level of hierarchy in Fig. 1 the deployment includes, among others, the binding between the interface $\mathsf{l_c}$ of **A** and the interface $\mathsf{l_p}$ of **B**.

3 Defining Correct Behaviour

Control (i.e., non-functional) operations can introduce changes on the component behaviour. For instance, adding or replacing a sub-component may add features (new actions) to the system. Since we made the assumption (this is a restriction

with respect to the Fractal specification) that no functional operation can fire control operations, then we are interested in three phases in the components behaviour:

1. Deployment: this is the building phase of a component. In this phase the component's content (its sub-components) is defined as well as the initial transformation phase (sequence of control operations), as described usually in the application ADL. The application deployment typically ends with a recursive start operation.
2. Running phase: only functional operations occur here.
3. Reconfiguration: we distinguish between non-structural reconfigurations (life cycle and binding controls) and structural transformations (adding, removing or updating components).

From these definitions, we discuss the *correctness* of the component system:

1. *Initial composition*: "Is the deployed system behaving correctly?". The concept of "correct behaviour" covers the absence of dead-locks and in general safety and liveness properties (common sense properties like not using an unbound required interface, or any user-requirement expressed as a temporal logic property). Ultimately, it could be "Does this implementation respect a pre-defined specification? (with respect to some implementation pre-order)".
2. *Reconfiguration*: "After a transformation phase, does the system behave correctly?". This covers both preservation of some properties valid before the transformation, and the satisfaction of a new set of properties, corresponding to features added by the transformation. These proofs must take into account the intricate interplay between functional and non-functional actions during transformation, like the management of the internal state of subcomponents. For example, one can expect to be able to prove the safety and transparency (from the user point of view) of the replacement of a components by another one.

We want to provide the user with tools that help answer those questions **before** deploying the application or applying a transformation, so he can be confident about the reconfigurations he will apply and therefore, have a reliable system.

4 Formalism

In [4] we have defined a parameterized and hierarchical model for synchronised networks of labelled transition systems. We have shown how this model can be used as an intermediate format to represent the behaviour of distributed Java applications, and check their temporal properties.

Our model is an adaptation of the *symbolic transition graphs with assignment* of [13] into the *synchronisation networks* of [2]: we extend the general notion of Labelled Transition Systems (LTS) and hierarchical networks of communicating systems (synchronisation networks) by adding parameters to the communication events in the spirit of [13].

We model the behaviour of a process as a parameterized Labelled Transition System (pLTS). We use parameters both for encoding data in value passing messages and for manipulating indexed families of processes.

Then we use a parameterized Net to synchronise a finite number of processes. A parameterized Synchronisation Network (pNet) is a form of generalised parallel operator, where each of its arguments is typed by a **Sort** that is the set of its possible observable actions.

The actions to be synchronised between the arguments of the Net are encoded in a *transducer* automaton. A state in the transducer defines a particular set of synchronisations, then a state change in the transducer introduces a new set of synchronisations, i.e. it models a dynamic reconfiguration. A Net with a unique state is called a *static* Net.

Given a finite instantiation of the parameters in the model, we have introduced in [4] an automatic procedure producing a (hierarchical) finite instantiation of the parameterized LTSs and Nets. Having done the instantiation, we can generate the synchronisation product, which is an LTS encoding the full behaviour of the Net when synchronising the actions of its processes as defined in the transducer. Since the synchronisation product is an LTS, it can be used as an argument in a upper Net definition. In other words, we do support hierarchical composition of processes.

Our formalism fits nicely in the components model. The behaviour of a primitive component is a LTS, that can be specified by the developer, or derived from code analysis. For a given composite, its content is the arguments of the Net and its initial bindings are encoded in the initial state of the transducer. The LTS of a composite encodes the functional behaviour of the component but also the control operations that do not change the geometry of the composite, namely start/stop, and bind/unbind operations. In the sequel, we define our transducers using a set of small automata, that we call *controllers*. On this model, we can check all properties during and after the "initial composition", and involving reconfigurations only relying on start, stop, bind, and unbind.

We deal with transformations that change the arity of the Net or the structure of the application (add/remove/update of components) as transformers of the model: starting with a hierarchical model in a given state, we build a new model after a sequence of transformations, in which we maintain the state of the components that were not changed by the transformation. We can then check for the properties (preserved or new) of the reconfigured system.

5 Building the Behaviour for the Example

We start building an automaton for each component encoding both its functional and non-functional (control) behaviours. In this section, we show how we build the controller automata, in a bottom-up fashion, for primitives and for composite components.

To benefit from the compositional properties of our models, we define this construction in the context of a given temporal logic formula, or more generally

for a given set of actions that the user wants to observe. Then we shall consider *abstract* automata for a given family of *hidden actions* (renamed as τ actions), or conversely for a given family of *visible actions* (all others are hidden), minimised by weak bisimulation at each step of the construction.

In particular, specific models can be constructed to exhibit to check the correct detection of some classes of errors.

5.1 Computing Controller Automata

We introduce a general purpose *Controller* expressed as a parameterized Network (pNet). In this general purpose Controller we define a finite number k of sub-component automata ($\mathbf{SubC^k}$), a life-cycle automaton ($\mathbf{LF}$), a finite number np of external ($\mathbf{E\_PI^{np}}$) and internal ($\mathbf{I\_PI^{np}}$) provides interface automata, and a finite number nr of external ($\mathbf{E\_RI^{nr}}$) and internal ($\mathbf{I\_RI^{nr}}$) requires interface automata. Synchronised actions are encoded by links between two or more processes.

To obtain the *Controller* for a component we instantiate the general controller based on its sub-components and interfaces. For instance, for the component $\mathbf{C}$,

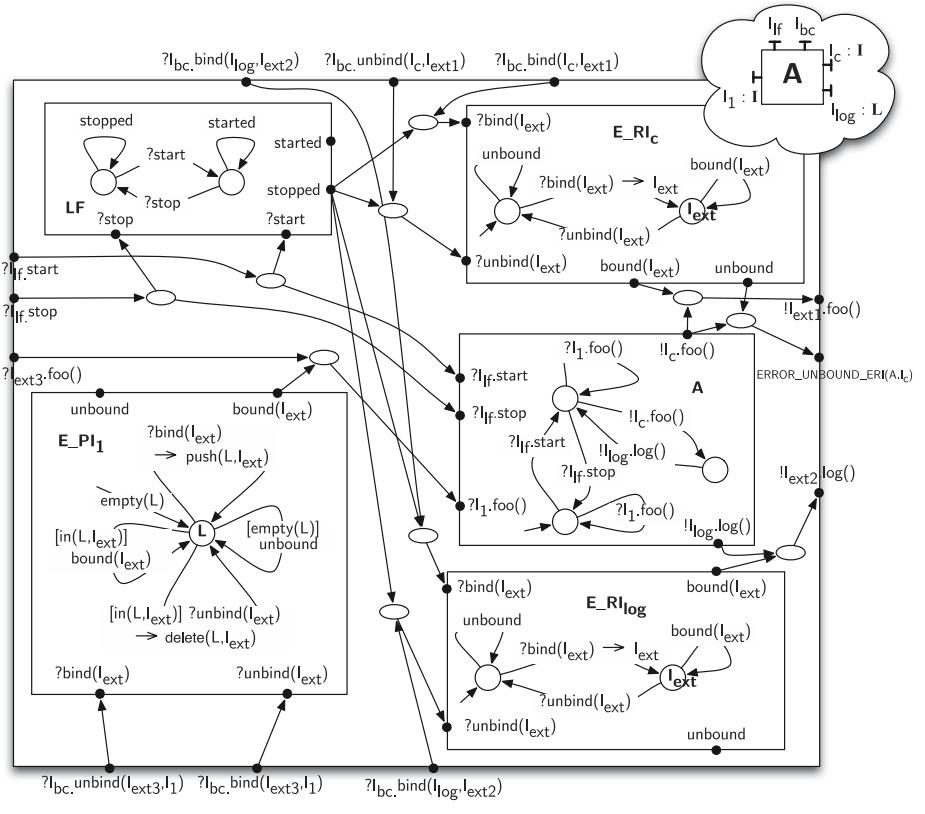


Fig. 2. Controller for $\mathbf{A}$

the set $\{\mathbf{SubC^k}\}$ becomes $\{\mathbf{A}, \mathbf{B}\}$. Since we build the controller automata in a bottom-up fashion, the controllers of the sub-components have been already built when we compose them. The automaton encoding the sub-component behaviour is the controller of this sub-components, hiding the internal functional actions that are not specified as visible.

Please remark that this instantiation fixes the set of sub-components and internal/external interfaces. The resulting pNet is still parameterized, and its actions contain variables for value-passing and for reference-passing.

For a primitive component, the set $\{\mathbf{SubC^k}\}$ is reduced to a single automaton which encodes its functional behaviour; the set of internal interfaces ($\{\mathbf{I\_PI^{np}}\}$ and $\{\mathbf{I\_RI^{nr}}\}$) is empty. The functional behaviour automaton encodes calls and receptions of methods on the component interfaces (in addition to internal actions). Whether it is obtained by source analysis or given by the user is outside the scope of this paper.

Because of space restrictions we only present the controller for component **A** which is the product of the 5 LTSs composing the pNet shown in Fig. 2 (we use an ellipse when more than two actions are synchronised). In [5] we describe the full example.

In Fig. 2, $\mathsf{I}_{\mathsf{ext}*}$ are variables encoding the set of external interfaces to which the interfaces of **A** can potentially be bound. This set is instantiated at the next level of hierarchy by type matching analysis. For instance when building the controller of **C**, the variable $\mathsf{I}_{\mathsf{ext1}}$ in the figure becomes the set $\{\mathsf{B}.\mathsf{I}_\mathsf{P}\}$.

In addition, the controller pieces in Fig. 2 include some of the constraints introduced in Section 2, e.g. that the bindings of requires interfaces are only possible when the component is stopped or that calls to requires interfaces are only possible when these interfaces are bound.

5.2 Detecting Errors

We can introduce in our model the detection of common sense errors (undesired behaviours) introduced in Section 3. For instance, by triggering an ERROR_UNBOUND message upon a call to the operations of the interface $\mathsf{I}_{\mathsf{log}}$ when it is unbound, we can detect the erroneous uses of the $\mathsf{I}_{\mathsf{log}}$ interface. This is shown in Fig. 3.

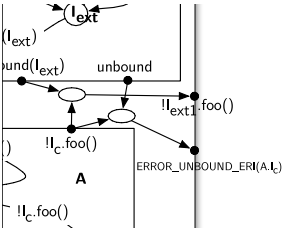


Fig. 3. Zoom into the **A** controller detecting errors

In addition to common sense errors, others undesired behaviours are directly or intrinsically defined in the Fractal specification. In order to keep simplicity and clarity during our guided example, we will consider only the error consisting in calling an operation on an unbound interface.

5.3 Species of Temporal Properties

All the temporal properties (that do not involve a structural transformation) can be expressed and verified directly on the controller automaton of a component, or of the whole application. Yet, it is possible to define classes of properties that can be checked on smaller systems, avoiding to build the global state-space. This section identifies abstractions and tools allowing to verify some specific categories of properties.

For a component **C** (including the full application itself), let us call $\mathbf{O}_F$ the set of external functional operations, $\mathbf{O}_E$ the set of observable errors, $\mathbf{O}_I$ the set of internal actions chosen to be observable. Then we define the set of deployment action as $\mathbf{O}_D = \neg(\mathbf{O}_F \cup \mathbf{O}_E \cup \mathbf{O}_I)$.

Deployment. As we mention in Section 2.2, a system is deployed in a bottom-up fashion in the component hierarchy. At each node (composite component) of the system a specific deployment is applied.

This deployment is defined by the user ; e.g. in Fractal, the bindings for the sub-components of a composite, can be given using its ADL. The deployment is a sequence of internal control operations of the composite, possibly interleaved with functional operations, and terminating with a distinguished successful state $\surd$. In our example (Fig. 1):

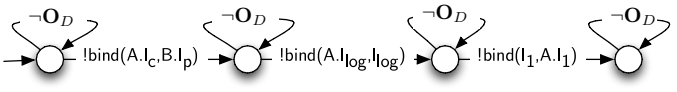


Fig. 4. Deployment automaton for **C**

The interplay between the building of all components of the application, and their start operations (that are usually applied recursively after building) may be quite complex and error-prone. So it may be useful for the developer to check, independently, that the deployment (possibly without start) of each component succeeds, and that the global deployment, including start operations, is also successful. This will be checked on the synchronisation of the component controllers with their respective deployment automata.

Functional behaviour. A functional property is a property concerning only functional actions, or more precisely properties of a system after correct deployment, on a system in which we forbid any subsequent control action. This kind of formulas can be model-checked on a controller automaton for which we already

have proved correct deployment, and in which we build only the relevant part of the behaviour, either by an had-hoc construction algorithm, or using on-the-fly techniques.

Functional behaviour properties are useful for component systems that do not perform any reconfiguration or for which non-functional actions have a transparent behaviour regarding functional aspects, i.e. non-functional actions commute with functional ones.

Non-structural Reconfiguration. Non-structural reconfiguration, i.e. involving only bind, unbind, start and stop operations, can be dealt with directly on the controller automaton. Indeed, the interleaving between functional and non-functional actions may have consequences on the state of the system ; we cannot provide any general abstraction fitting with this case that could reduce the complexity of the model construction for this class of properties.

Structural Transformations. *Remove*, *add* and *update* are the main control operations that modify the content of a composite. The first remark is that there is no hope to encode all possible future transformations in the model. Then, technically, *add* and *remove* operations change the arity of the enclosing Net, so they cannot be modelled as transducer transitions. Instead we model the structural transformation operations as functions transforming the whole hierarchical model of the application ; each elementary structural change affects a single Net or LTS in the model.

Update could be expressed as a sequence *unbind*;remove;add;bind**, but this would lead both to less efficient implementations and to more complex model constructions and proofs: we are interested in expressing full sequences of transformations, that preserve properties of the system, while elementary transformations usually don't.

The main difficulty with structural reconfigurations is that one wants to keep the rest of the system in the same state. A large application should not be stopped when updating or adding a specific sub-component, and the state of a replaced component itself should be preserved whenever possible. The framework ensures minimum conditions before replacements (in terms of stopped/unbound state), but we have to assume that the developer will specify which data from the replaced components are to be saved, and how this data will be mapped in the new component.

In our formalism, this tree transformation and state transfer is expressed on the hierarchical pNets, as the following sequence of steps :

- build the new hierarchical pNet, by replacement of the transformed part; call $\mathcal{S}'$ the semantics of this new system;
- define a mapping between actions in the original and the new systems, based on a user-defined mapping between the action names and parameters in the replaced component;
- identify the set $\mathcal{T}$ of states on the initial system where the transformation is possible;

- build a synchronised product of the old and new system, using the mapping of old to new actions, and adding in each state of $\mathcal{T}$ a transition $\xrightarrow{t}$ encoding the transformation; we call $\mathcal{T}'$ the image of $\xrightarrow{t}$ in this product;
- finally obtain the controller automaton of the transformed system, $\mathcal{A}'$ defined by: the set of initial states of $\mathcal{A}'$ is $\mathcal{T}'$, the states and the transitions of $\mathcal{A}'$ are those of $\mathcal{S}'$ reachable from $\mathcal{T}'$.

The actions mapping will eventually be defined in terms of the source language of the application, but this is out of the scope of this paper.

6 Proving Properties

In our tools, we use the modal μ-calculus as a powerful internal language for logic formulas. For this paper, we prefer to use an Action-based Computation Tree Logics (ACTL, see e.g. [9]), that may be more suitable for a human reader.

1. **Deployment:** We want at first to verify that the deployment for a component is always successful. This is done by proving the ACTL formula
$$[true]\surd \qquad (1)\text{ (all paths lead to success)}$$
in the synchronisation product between the component controller and its deployment. This formula is true for the deployment in Fig. 4.
A second property we would like to verify is the absence of error during the deployment. This is done by proving the formula
$$\mathbf{EF}_{\neg \mathbf{O}_E} < \mathbf{O}_E > false \qquad (2)$$
in the synchronisation product between the component controller and its deployment. This property is also true for the deployment in Fig. 4. However, in a scenario very reasonable, let's suppose the user starts the component **C** at the end of the deploy (which means to add a !start transition before the state $\surd$). Under this scenario the property is not true anymore (even though the deployment is possible), and the model-checking tool give us the counter-example shown in Fig. 5

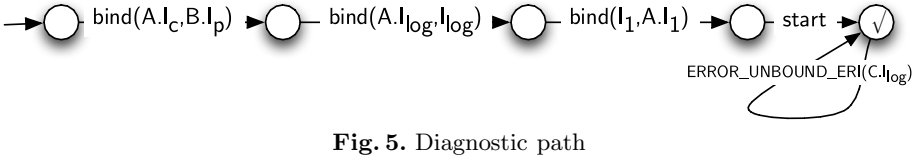


Fig. 5. Diagnostic path

The error is because the required interface $\mathsf{C.I_{log}}$ may be used before it is bound, which in fact is true since the interface $\mathsf{I_{log}}$ of **C** will be bound at the next level of hierarchy (when deploying **System**). This example also shows us the importance of the hierarchical behaviour of start and stop.

2. **Functional behaviour:** We would like to verify the absence of errors during a running phase, i.e. the absence of errors after the deploy until a new reconfiguration phase. If $\mathbf{O}_E$ is the set of observable errors and $\mathbf{O}_D$ is set of observable control operations, then we can verify the property in **System** by proving the ACTL formula:

$$\mathbf{A}[true]\sqrt{\mathbf{AG}_{\neg \mathbf{O}_D}[\mathbf{O}_E]false} \tag{3}$$

   is true in its controller. The proof is successful for **System**.
3. **Transformation:** Suppose we do, during the application running-phase, an update of the sub-component **B** in **C** by a component **B2**. **B2** has a similar behaviour than **B**, but in addition it logs the calls to its I_p interface using its I_log interface. When we prove formula (3), it becomes false in this updated system, and the tool gives us a path containing the action ERROR_UNBOUND_ERI(B2.log). This is because in the initial deployment of the system, we did not bind the interface I_log of **B**. Since **B** did not use its interface I_log, the composition did not produced an undesired behaviour. However, the new **B2** uses its I_log interface, and so it produces the error. So the update of **B** by **B2** should be followed by a binding of its I_log interface. This example, likely to happen in real systems, shows the necessity of formal tools of verification for checking reconfiguration requirements.
   If after the update and before starting the system, we bind the interface I_log of **B2** to the internal interface I_log of **C**, then the property is preserved.

6.1 Tools

Figure 6 shows the ADL description file for the upper level of hierarchy. In line 17 and 18 we suggest a way to introduce functional behavioural specification of primitive components.

System.fractal

```
1  <?xml version="1.0" encoding="ISO-8859-1" ?>
2  <!DOCTYPE .... >
3
4  <definition name="components.System">
5
6     <component name="C"
7           definition="components.C">
8        <interface name="log" role="client"
9              signature="components.LogInterface"/>
10       <interface name="l1" role="server"
11             signature="components.I1Interface"/>
12    </component>
13
14    <component name="Logger">
15       <interface name="log" role="server"
16             signature="components.LogInterface"/>
17       <content class="components.LoggerImpl">
18          <behaviour file="LoggerBehav"
19                format="Aldebaran"/>
20       </content>
21    </component>
22
23    <binding client="C.log"
24          server="Logger.log"/>
25 </definition>
```

Fig. 6. System ADL

Component	Controller	Static Aut.
A	24/99	24/91
B	16/98	16/90
Logger	4/16	4/14
C	432/2168	12/39
System	36/151	6/19
B2	24/107	24/99
C {update(B,B2)}	1786/7082	20/58

Fig. 7. Branching reduced automata sizes (states/transitions) of the example

We developed a tool prototype in Java which takes as inputs the system ADL and the functional behaviour of primitives to automatically generate the models described in this paper. We use the CADP [10] tool-set to do the synchronisation product and the model-checking of formulas. From the ADL description, our tool prototype generates, once instanced, the synchronisation product in Exp-V2 format (and ASCII list of synchronisation vectors). The automaton describing the functional behaviour of primitives is taken directly from its file (line 18 in Fig. 6). In our example the primitive automata are in Aldebaran format (a simple ASCII representation of finite LTS). Our tool also generates a script to build the system (svl script from CADP). Finally the proofs are verify using evaluator, an on-the-fly model checking tool included in CADP. Figure 7 shows some results for the generated automata in our example; the CADP tool-set allows us to handle systems with as much as 100 millions states at each level of the construction.

7 Related Work

Most component frameworks available today only have tools for checking the static type compatibility of interfaces. Work on behaviour compatibility is quite recent, and not yet available on industrial plate-forms. We mention here research works and tools that may be the closest to our approach.

Wright [1] was an early proposal for specifying the behaviours of components in an Architecture Description Language (ADL). They use connectors similar to our Nets, that define the possible interactions between a set of roles (specification of sub-components). The behaviours of roles is specified in CSP, and they have a notion of compatibility based on a variant of CSP's refinement pre-order that ensures the absence of deadlock.

Darwin [11] is an Architecture Description Language, in which a distributed program is represented as a hierarchical composition of subsystems, with interacting processes at the leaves of the hierarchy. The behaviours are specified and computed in a way similar to ours, including weak bisimulation minimisation during the bottom-up construction. Verification of safety and liveness properties, specified in terms of finite-state automata, is done by the Tracta tool. The main difference with our approach is that Darwin expresses only the functional operations of the components, and does not support system reconfiguration.

Sofa [14] defines a hierarchical component system. At each level of hierarchy, a *frame protocol* specifies the external behaviour of the component, while a *architecture protocol* describes an implementation capturing also the internal synchronisation between its sub-components. The architecture protocol can be automatically generated from the frame protocols of its sub-components. The behaviours are expressed as regular expressions, and the substitutability of components is based on trace language inclusion, though it is yet unclear how to compare with our bisimulation-based semantics. One specificity of Sofa is its sophisticated mechanism for detecting and reporting errors. They also have a syn-

tax for expressing asynchronous operations, for which the emission of a method call and the return of its result may be interleaved with other events.

A quite different approach is advocated by Carrez, Fantechi and Najm in [8]. They propose a (non-hierarchical) component model in which interfaces are given a behavioural type expressed in a kind of modal process algebra. Then, they define the *sound assembly* of components as the conjunction of compliance of components to their interface (contracts), and compatibility between interfaces. The type language definition ensures that the compatibility is decidable and can be computed efficiently. Unfortunately, the compliance relation is more complex, and may even require theorem-proving techniques, but only needs to be guarantied once for a given component.

Cadena [12] is a development and verification environment for building real time systems using CORBA. They extend the Interface Definition Language to add light-weight specification of component behaviour and dependencies using a BIR-like language. Then they use Bogor, a specialised model-checking tool for Cadena, to verify properties expressed using logical patterns. Cadena does not work with hierarchical components, and the model-checking tool lacks compositionality. They assume a correct deployment and do not support dynamicity.

8 Discussion and Conclusion

This paper provides methods and tools allowing the user to prove the correctness of the behaviour of hierarchical components. One of our main contributions is the specification of the behaviour of non-functional aspects, and the hierarchical building of LTSs modelling the behaviour of the system of components. Our approach rely on the definition of a generic controller allowing (once instantiated) to encode the whole behaviour of any component except non-structural reconfiguration. Then a component behaviour is obtained by synchronisation product of the LTSs expressing the behaviour of its content and the control behaviour associated to its interfaces. Structural (dynamic) reconfiguration is handled by a LTS transformation. The tools provided to the user include:

- a controller automaton allowing to prove general properties on the behaviour of a component provided no structural reconfiguration is considered;
- an error detection: firing of error messages upon common sense errors can automatically be added; then, for example, the user may prove the absence of such messages in order to assert the correctness of the application;
- modelling of structural reconfigurations as transformations of the application model, thus allowing to reason about the most general components reconfigurations.

A promising perspective is to extend this framework in order to specify and verify the behaviour of asynchronous distributed components. Such a work would benefit from our previous experience in specification of asynchronous communicating objects [3], and consist in extending and adapting the notion of asynchronous method calls, request queues, etc.

Finally, many approaches are being developed to cover the right composition of components considering their functional aspects. One of the strongest advantage of using components is the separation of concerns from the user point of view. However, when coming to behavioural verification, one still needs to take into account the inter-play between functional and non-functional aspects, at least for existing component models. The main contribution of this paper is to encode the deployment and reconfigurations as part of the behaviour of the system, and thus verify the behaviour of the whole component system.

References

1. R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.
2. A. Arnold. *Finite transition systems. Semantics of communicating sytems.* Prentice-Hall, 1994.
3. I. Attali, T. Barros, and E. Madelaine. Formalisation and proofs of the chilean electronic invoices system. In *XXIV International Conference of the Chilean Computer Science Society (SCCC 2004)*, pages 14–25, Arica, Chili, November 2004. IEEE Computer Society.
4. T. Barros, R. Boulifa, and E. Madelaine. Parameterized models for distributed java objects. In *Forte'04 conference*, Madrid, 2004. LNCS 3235, Spinger Verlag.
5. T. Barros, Henrio Ludovic, and E. Madelaine. Behavioural models for hierarchical components. Technical Report RR-5591, INRIA, June 2005.
6. Francoise Baude, Denis Caromel, and Matthieu Morel. From distributed objects to hierarchical grid components. In *International Symposium on Distributed Objects and Applications (DOA), Catania, Sicily, Italy, 3-7 November*. LNCS, 2003.
7. E. Bruneton, T. Coupaye, and J. Stefani. Recursive and dynamic software composition with sharing. Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02), June 2002.
8. A. Fantechi C. Carrez and E. Najm. Behavioural contracts for a sound assembly of components. In Springer-Verlag, editor, *in proceedings of FORTE'03*, volume LNCS 2767, November 2003.
9. R. De Nicola and F.W. Vaandrager. Action versus state based logics for transition systems. In I. Guessarian, editor, *Semantics of Systems of Concurrent Processes*, volume 469 of *LNCS*, La Roche Posay, France, 1990. Springer.
10. H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. *European Association for Software Science and Technology (EASST) Newsletter*, 4:13–24, August 2002.
11. D. Giannakopoulou, J. Kramer, and S. Chi Cheung. Behaviour analysis of distributed systems using the tracta approach. *Automated Software Engg.*, 6(1):7–35, 1999.
12. J. Hatcliff, X. Deng, M. B. Dwyer, G. Jung, and V. P. Ranganath. Cadena: an integrated development, analysis, and verification environment for component-based systems. In *ICSE '03 conference*, pages 160–173, Washington, DC, USA, 2003. IEEE Computer Society.
13. H. Lin. Symbolic transition graph with assignment. In U. Montanari and V. Sassone, editors, *CONCUR '96*, Pisa, Italy, 26–29 August 1996. LNCS 1119.
14. F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11), nov 2002.

On-the-Fly Emptiness Checks for Generalized Büchi Automata

Jean-Michel Couvreur[1], Alexandre Duret-Lutz[2], and Denis Poitrenaud[2]

[1] LaBRI, Université de Bordeaux I, Talence, France
[2] LIP6, Université de Paris 6, France

Abstract. *Emptiness check* is a key operation in the automata-theoretic approach to LTL verification. However, it is usually done on Büchi automata with a single acceptance condition. We review existing on-the-fly emptiness-check algorithms for *generalized* Büchi automata (i.e., with *multiple acceptance conditions*) and show how they compete favorably with emptiness-checks for degeneralized automata, especially in presence of weak fairness assumptions. We also introduce a new emptiness-check algorithm, some heuristics to improve existing checks, and propose algorithms to compute accepting runs in the case of multiple acceptance conditions.

1 Introduction

The automata-theoretic approach to model-checking [22] uses automata on infinite words to represent a system as well as a formula to check on this system. Both automata are synchronized, and a key operation is to determine whether the resulting automaton is empty (i.e., contains no accepting run). This operation is called *emptiness check*. An on-the-fly emptiness check allows the synchronized automata to be constructed lazily while it runs. This is a win if the emptiness check answers before the whole synchronized product is completed.

We follow up on a paper by Schwoon and Esparza [17] who compared two classes of on-the-fly emptiness checks: those based on nested depth-first searches (NDFSs) versus those computing strongly connected components (SCCs). Their measures for Büchi automata with single acceptance conditions led to the following conclusions:

- Couvreur [3]'s algorithm is the best at computing accepting SCCs,
- Schwoon and Esparza [17]'s algorithm is the best of NDFS-based checks,
- for weak Büchi automata [1], a simple DFS is enough; otherwise SCC-based algorithms should be preferred to NDFSs unless bit-state hashing is used.

Here we explore these algorithms on Büchi automata with multiple acceptance conditions (the so-called *generalized Büchi automata*) to stress the advantages of generalized emptiness checks over traditional algorithms.

Section 2 introduces the emptiness-check problem and existing algorithms. Section 3 describes our experimental workbench. The later two sections present some contributions to each class of algorithms as well as algorithms for the computation of accepting runs.

P. Godefroid (Ed.): SPIN 2005, LNCS 3639, pp. 169–184, 2005.

2 Emptiness Check

2.1 Transition-Based Generalized Büchi Automata

A *Transition-based Generalized Büchi Automaton* (TGBA) over the alphabet Σ is a Büchi automaton with labels on transitions, and generalized acceptance conditions on transitions too. It can be defined as a tuple $A = \langle \Sigma, \mathcal{Q}, \mathcal{F}, q^0, \delta \rangle$ where

- Σ is an alphabet,
- $\mathcal{Q}$ is a finite set of elements called *states*,
- $\mathcal{F}$ is a finite set of elements called *acceptance conditions*,
- $q^0 \in \mathcal{Q}$ is a distinguished initial state,
- $\delta \subseteq \mathcal{Q} \times (2^\Sigma \setminus \{\emptyset\}) \times 2^\mathcal{F} \times \mathcal{Q}$ is the transition relation, where each transition is labeled by a nonempty set of letters of Σ and a set of acceptance conditions of $\mathcal{F}$.

A *run* of A is an infinite sequence $\langle q_0, l_0, f_0, q_1 \rangle \langle q_1, l_1, f_1, q_2 \rangle \cdots \langle q_j, l_j, f_j, q_{j+1} \rangle \cdots$ of transitions of δ, starting at $q_0 = q^0$. Such a run is said to be *accepting* if $\forall f \in \mathcal{F}$, $\forall i \geqslant 0$, $\exists j \geqslant i$, such that $f \in f_j$, i.e., if its transitions are labeled by each acceptance condition infinitely often.

An *emptiness check* is an algorithm that tells whether at least one accepting run exists. On a TGBA, it amounts to testing whether there exists a circuit that (1) is accessible from q^0, and (2) is labeled by all acceptance conditions $\mathcal{F}$.

The l_is can be used to describe words recognized by a run, but for the purpose of finding accepting runs we shall not be concerned by l_is and Σ. Also note that we use acceptance conditions as labels on transitions, rather than as the usual sets of transitions, because that is how it is coded in practice.

Other Büchi automata in use for model-checking, have acceptance conditions on states rather than transitions, and are often not generalized (i.e., $|\mathcal{F}| \leqslant 1$). While the benefit of TGBAs in the process of translating LTL formulæ is already quite clear [3, 10, 4], few people are actually using them for emptiness-check, because mainstream algorithms work on non-generalized, state-based, Büchi automata.

A *degeneralization* is the transformation of an automaton with $|\mathcal{F}| > 1$ into an automaton with $|\mathcal{F}| = 1$ [10]. This operation may multiply the size of the automaton by at most $|\mathcal{F}|$ to produce a transition-based automaton, and by at most $|\mathcal{F}| + 1$ to produce a state-based automaton. Such a blowup is often disregarded when only the automaton that represents the property needs to be degeneralized: such automata are usually small. Acceptance conditions can also be used to express some class of fairness constraints such as *weak fairness*. In Spin, weak fairness is handled using a degeneralization algorithm [13, p. 182]. As we shall see in our measures, the degeneralization is much more painful when applied to weak fairness.

2.2 Existing Algorithms

Two classes of on-the-fly emptiness-check algorithms exist: nested depth-first searches (NDFSs), and algorithms that compute strongly connected components (SCCs). Fig. 1 shows how the algorithms we cite relate to each other.

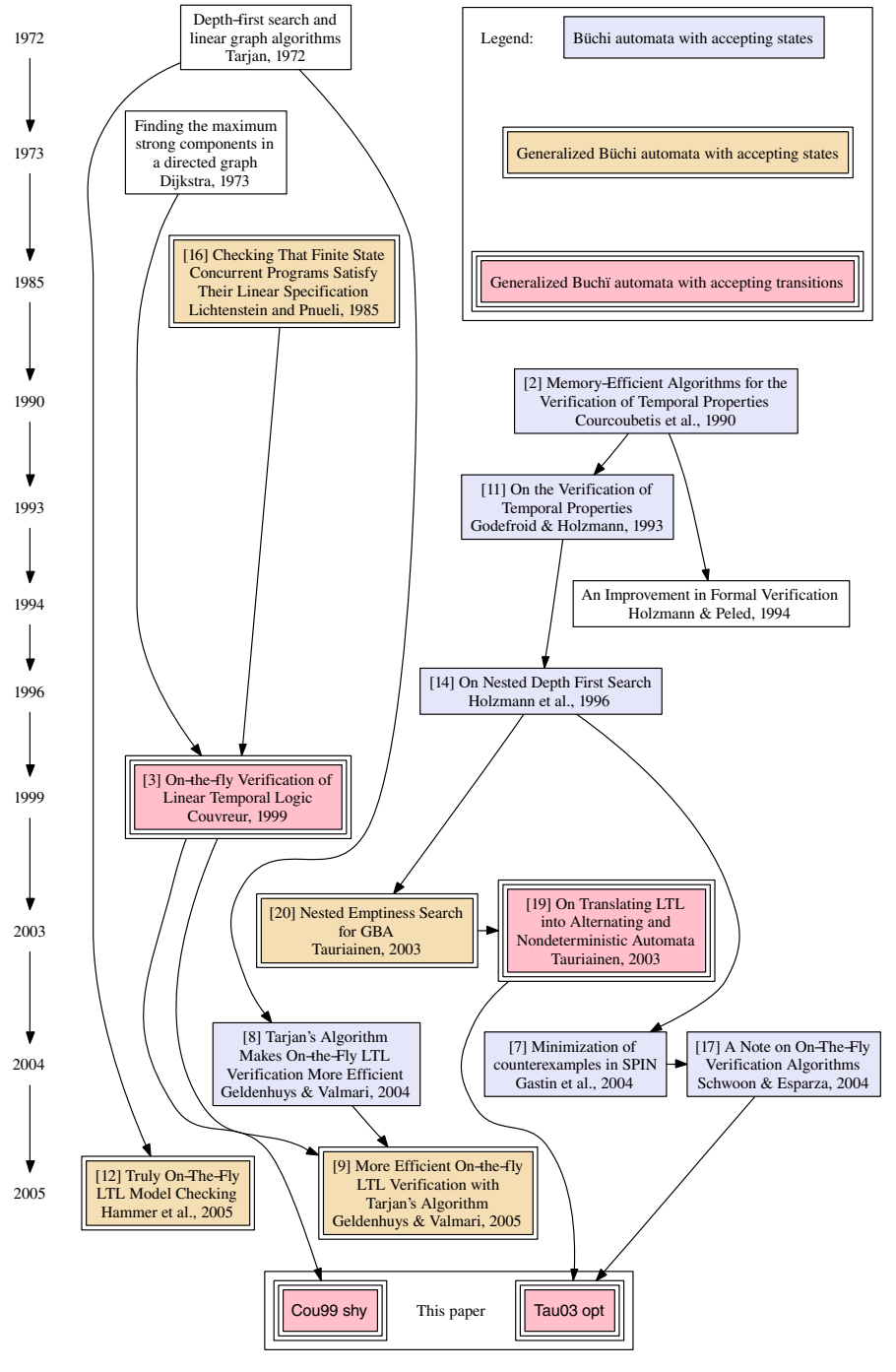


Fig. 1. A family tree of emptiness-check algorithms

```
c1  // Let ⟨Σ, Q, δ, q⁰, F⟩ be the
c2  // input automaton to check.
c3  todo: stack of ⟨state ∈ Q, succ ⊆ δ⟩
c4  SCC: stack of ⟨root ∈ ℕ, la ⊆ F,
c5                 acc ⊆ F, rem ⊆ Q⟩
c6  H: map of Q ↦ ℕ
c7  max ← 0
c8
c9  main():
c10   push(∅, q⁰)
c11   while ¬todo.empty()
c12     if todo.top().succ = ∅
c13       pop()
c14     else
c15       pick one ⟨_, _, a, d⟩ off todo.top().succ
c16       if d ∉H
c17         push(a, d)
c18       else if H[d] > 0
c19         if merge(a, H[d]) = F
c20           return ⊥
c21   return ⊤

c23 push(a ⊆ F, q ∈ Q):
c24   max ← max + 1
c25   H[q] ← max
c26   SCC.push(⟨max, a, ∅, ∅⟩)
c27   todo.push(⟨q, {⟨s, l, a, d⟩ ∈ δ | s = q}⟩)
c28
c29 pop():
c30   ⟨q, _⟩ ← todo.pop()
c31   SCC.top().rem.insert(q)
c32   if H[q] = SCC.top().root
c33     forall s ∈ SCC.top().rem
c34       H[s] ← 0
c35     SCC.pop()
c36
c37 merge(a ⊆ F, t ∈ ℕ):
c38   r ← ∅
c39   while (t < SCC.top().root)
c40     a ← (a ∪ SCC.top().acc
c41           ∪ SCC.top().la)
c42     r ← r ∪ SCC.top().rem
c43     SCC.pop()
c44   SCC.top().acc ← SCC.top().acc ∪ a
c45   SCC.top().rem ← SCC.top().rem ∪ r
c46   return SCC.top().acc
```

Fig. 2. Another presentation of the algorithm of Couvreur [3] to check the emptiness of TGBAs

Nested Depth-First Searches. NDFSs were initially developed for Büchi automata with only one acceptance condition for states [2]. Basically, a NDFS will perform a first DFS rooted at q^0 until it finds an accepting state s, and from there starts a second DFS to check whether s is reachable from itself. This naive algorithm was then further refined so that both DFSs could share the same hash table [11], to exit earlier and to support partial order reductions [14].

Holzmann et al. [14]'s algorithm has been refined by Gastin et al. [7] and Schwoon and Esparza [17]. In parallel, Tauriainen generalized it to support multiple acceptance conditions on states [20], or transitions [19]. Switching from states to transitions is easy; the real challenge was to devise a way to handle generalized acceptance conditions. Tauriainen did this by repeating the inner DFS several times (at worst $|\mathcal{F}|$ times).

Strongly Connected Components. Another strategy is to compute the maximal strongly connected components (MSCCs) of the automaton. Let us define a *trivial* SCC as a single state without self-loop. If the union of all the acceptance conditions occurring in a non-trivial SCC is $\mathcal{F}$, and that SCC is accessible from q^0, then one can assert the existence of such an accepting run. This is the essence of the algorithms of Couvreur [3], Geldenhuys and Valmari [8, 9], and Hammer et al. [12].

We present an iterative version of Couvreur [3]'s algorithm in Fig. 2 in order to introduce two heuristics in Section 4.1. This algorithm is based on the fact that any

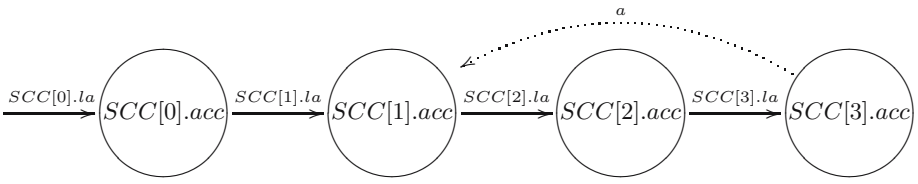


Fig. 3. The meaning of la and acc in SCC

graph contains at least one MSCC without outgoing arc. To list all MSCCs, one should find such a terminal MSCC, remove it from the graph, and then list all MSCCs of the resulting graph [16]. It turns out this requires to visit each transition only once.

To do so the algorithm explores the graph in depth-first order. $todo$ is a DFS stack, on which each item contains a state and the set of its successors that have yet to be visited. (In practice this set of successors may not need to be represented explicitly and would be replaced by the necessary information to compute the next successor of the state.) H maps each state to its rank in the depth-first order, and $H[q] = 0$ indicates that q belongs to a removed MSCC.

During the DFS, a chain of SCCs is maintained as a stack, SCC, depicted on Fig. 3. To each SCC is associated the rank of the first state of the SCC ($root$), the union of acceptance conditions in the SCC (acc), the acceptance conditions labeling the transition coming from the previous SCC (la), and the list of states of the SCC that have been fully explored (rem). ($SCC[0].la = \emptyset$ by convention and is never used.) Using this structure, two visited states q_1 and q_2 belong to the same SCC if $\max\{r \mid SCC[r].root \leqslant H[q_1]\} = \max\{r \mid SCC[r].root \leqslant H[q_2]\}$.

Initially, each new state is pushed on the stack as a trivial SCC with an empty acc (line c26). When the DFS reaches a successor q that has already been visited and has not been removed (line c18), all SCCs between the SCC to which q belongs and the top SCC (source of the transition) are merged into a single SCC. On the example of Fig. 3 where a back arc is found between $SCC[3]$ and $SCC[1]$, the last three SCCs would be merged into a single one with acceptance conditions $SCC[1].acc \cup SCC[2].la \cup SCC[2].acc \cup SCC[3].la \cup SCC[3].acc \cup a$. If that union is $\mathcal{F}$, then an accessible, non-trivial, and accepting SCC exists, and the algorithm reports $\bot$ (the automaton is not empty).

When the root of an SCC is popped (tested line c32), the SCC is known to be maximal and not accepting, so it can be discarded. The use of rem line c31 to remove the states of the MSCC line c33 could be avoided because when line c33 is reached, rem contains all the states s accessible from q (ignoring those with $H[s] = 0$), as the original algorithm did [3]. The current implementation favors run-time to memory consumption, indeed a concern from Schwoon and Esparza [17] was that computing transitions can be expensive. To be fair we will account for the size of rem in our measures of the stack size. (Geldenhuys and Valmari [9] provide alternative structures that address the same problem.)

Another SCC-based algorithm, Geldenhuys and Valmari [8]'s, has a similar handling of its stack: it keeps all states of partial SCCs, so it can remove them easily.

However it also stores an additional integer for each state ($lowlink$) that we will not account for in our measures. This algorithm works only on degeneralized automata.

Hammer et al. [12]'s algorithm is presented as an emptiness check for Linear Weak Alternating Automata (LWAA). However their algorithm translates an LWAA into a generalized Büchi automata on-the-fly during the emptiness check. The translation from LWAA could be coupled with any other emptiness-check algorithm presented here. The real part of their emptiness check follows the same logic as Couvreur's algorithm except it merges SCCs one by one while popping instead of immediately when a loop is found. It will therefore find an accepting SCC later than the algorithm of Fig. 2, only when this SCC is popped.

3 Experimentations

In this section we introduce the experimental framework in which we compare the aforementioned algorithms, and comment on the results. All the algorithms we use are implemented in the Spot library [4]. The random graph and random LTL formulæ generation algorithms are comparable to those presented by Tauriainen [21]. Of the 8 emptiness-check algorithms we compare, the first 4 are SCC-based: Cou99, is the algorithm of Fig. 2, Cou99 shy- and Cou99 shy are two variants of Cou99 described in Section 4.1, and GV04 is the algorithm of Geldenhuys and Valmari [8]. The other 4 are NDFS algorithms: CVWY90 [2], SE05 [17], Tau03 [19], and Tau03 opt (a variant of Tau03 presented Section 4.2). In tables, "×" indicates new algorithms that will be discussed in Section 4.

Because all our tests use TGBAs as input, we had to adjust CVWY90, GV04, and SE05 to handle transition-based automata (this is straightforward) and because they will not handle generalized acceptance conditions we also had to degeneralize the input automata for these 3 algorithms. (Hence the input can be $|\mathcal{F}|$ times larger.)

We exercised these algorithms on random graphs and concrete models, following a pattern similar to that of Geldenhuys and Valmari [8]. First we use them to check random graphs against LTL formulæ. Then we try them on two real models (the first of which also comes from Geldenhuys and Valmari [8]).

Table 1 presents our results when checking random graphs with all algorithms in 12 different setups. Each setup differs in how the graph and formulæ are generated. The random graphs have 1024 states and are generated with 3 different densities d of transitions (all 1024 states are accessible and the arity of each state follows a normal distribution with mean $1+1023d$ and variance $1023d(1-d)$). In columns headed "fair", transitions in the graph are additionally randomly labeled with 3 acceptance conditions to mimic weak fairness constraints; in the other columns the acceptance conditions, if any, will come only from the LTL formulæ.

The values presented for each experiment are means. They where computed by running each emptiness check on a set of 1300–3000 products generated as follows.

For each setup we consider 15 random graphs. On setups with random formulæ, each graph is checked against 200 LTL formulæ (converted to TGBAs using the algorithm of Couvreur [3]), yielding 3000 different products. On setups with "Human-generated formulæ", each graph is checked against 94 formulæ (and their negation) selected from the literature [5, 6, 18], yielding 2820 products.

In this first test we discarded all products without accepting run (i.e., keeping those where algorithms need not visit the whole automaton). For each setup the number of non-empty products remaining is written in italics on the same line as the density. For Tau03 we also discarded products with no acceptance conditions at all, because Tau03 is not designed to handle them; the resulting number of products is put in parentheses.

For each check of a non-empty product, we compute the ratios between (1) the number of distinct states visited and the number of states in the product TGBA, (2) the number of traversed transitions (a same transition can be accounted more than once) and the number of transitions of the product TGBA, and (3) the maximal size of the stack and the number of states of the product. For all algorithms, even if a degeneralization is required, ratios are computed against the product before any degeneralization. The table displays the means of each of these three ratios in %.

Our computation of the stack size deserves more explanations as not all algorithms use similar stacks. For all NDFS algorithms, we simply counted the number of states in the DFS stack. For Cou99, we counted the number of entries in $todo$ (its DFS stack), plus the size of rem for each of entry on SCC (this is because $succ$ can be represented as an iterator of constant size, and if you omit its rem field the size of SCC is bounded by that of $todo$). For GV04 we counted all items on $stack$ [8] (this is proportional to all states that are in the current SCC chain).

The algorithms presented here have a runtime proportional to the number of transitions explored. So the second value of each triplet allows to compare the runtimes. Also, for a fixed $|\mathcal{F}|$, the memory consumption of the algorithm is a linear combination of the number of states explored (first value) and of the size of the stack (second value).

A first remark concerns the results presented by Geldenhuys and Valmari [8], who compared their implementations of GV04 and CVWY90 using the same procedure (at the exception of the "fair" columns). We could not reproduce the important contrast they show between these two algorithms (neither could Hammer et al. [12]). For example, the 94 formulæ (and their negation) from the literature have been checked against 15 random graphs with a transition probability of 0.001. 2308 of the 2820 generated products are non-empty and GV04 has reported an accepting run after exploring an average of 7.5% of the states when CVWY90 needs to visit 7.7%. Geldenhuys and Valmari [8] report a rate of 8.99% for GV04 against 40.21% for CVWY90. These discrepancies are likely due to different parameters of the random graph generator.

The results for setups with randomly-generated formulæ are comparable to those with non-random formulæ; if anything, this only shows that random formulæ are not biased. Similarly, the density d does not seem to affect the algorithms much. Therefore it is much more interesting to compare the behaviors of the algorithms when acceptance conditions comes only from the formulæ or when additional acceptance conditions have been injected into the random graphs. In the former most TGBAs have few acceptance conditions (e.g., for formulæ from the literature, 40% of TGBAs have 0 or 1 acceptance condition, 40% have 2, and 20% have between 3 and 6), consequently the difference between CVWY90, GV04, SE05 (which require degeneralized automata) and the other algorithms are not striking. However on the "fair" setups, SCC-based algorithms often outrank NDFS ones. The poor results of Tau03 are mostly due to the logic of the origi-

Table 1. Comparison of algorithms for random graphs and random and real LTL formulæ

		Random formulæ						Human-generated formulæ					
	Algorithm	formula's cond.			fair			formula's cond.			fair		
	$d = 0.001$	*2328 (1318)*			*2188*			*2308 (2127)*			*1951*		
degeneralized	Cou99	6.8	4.5	4.5	18.1	11.1	13.8	7.4	5.1	4.0	16.3	10.6	10.4
×	Cou99 shy-	5.4	5.8	7.5	16.5	17.7	25.6	6.5	6.8	7.7	15.2	15.7	19.6
×	Cou99 shy	5.4	5.8	7.4	15.6	16.7	23.5	6.3	6.5	7.0	14.5	15.0	18.0
degeneralized	GV04	6.8	4.5	4.5	28.4	17.1	21.6	7.5	5.1	4.0	25.9	16.5	16.1
degeneralized	CVWY90	6.8	7.1	6.4	61.7	73.9	66.6	7.7	7.9	5.2	53.6	65.0	49.3
degeneralized	SE05	6.8	5.7	4.5	59.4	39.1	38.4	7.6	6.8	3.9	50.9	34.7	28.1
	Tau03	9.9	16.1	10.8	64.7	295.9	49.6	9.5	17.4	8.1	53.9	265.5	36.2
×	Tau03 opt	6.8	5.2	4.5	18.5	27.1	15.4	7.4	8.1	3.8	16.4	31.8	11.3
	$d = 0.002$	*2716 (1488)*			*2695*			*2569 (2304)*			*2548*		
	Cou99	4.8	2.2	3.1	17.5	8.5	11.4	4.5	2.6	2.4	13.7	7.4	7.6
×	Cou99 shy-	3.4	3.4	6.9	15.4	15.8	30.1	3.6	3.8	6.1	12.5	12.1	19.8
×	Cou99 shy	3.4	3.4	6.8	14.3	14.6	26.5	3.4	3.6	5.4	11.9	11.4	17.3
	GV04	4.8	2.2	3.1	29.1	14.1	18.5	4.6	2.8	2.4	23.2	12.8	11.9
	CVWY90	4.9	3.6	4.5	60.3	58.0	59.5	4.8	4.2	3.4	45.1	43.9	35.6
	SE05	4.8	2.8	3.1	56.8	30.2	32.9	4.7	3.5	2.4	42.0	24.3	19.8
	Tau03	8.5	12.5	9.1	61.3	265.5	46.5	7.1	12.5	6.3	40.9	185.4	27.3
×	Tau03 opt	4.8	2.7	3.1	17.8	23.9	12.7	4.5	4.5	2.4	13.9	26.3	8.3
	$d = 0.01$	*2978 (1569)*			*2979*			*2766 (2441)*			*2765*		
	Cou99	3.5	0.7	2.4	12.3	1.9	8.1	2.6	0.7	1.4	7.8	1.5	4.8
×	Cou99 shy-	1.7	1.5	11.7	8.2	8.3	66.6	1.6	1.6	10.6	5.6	5.4	39.2
×	Cou99 shy	1.6	1.5	10.7	6.9	7.0	53.0	1.4	1.3	7.7	4.8	4.4	29.9
	GV04	3.5	0.7	2.4	20.7	3.5	13.2	2.6	0.8	1.4	13.5	2.8	7.7
	CVWY90	3.6	1.1	3.3	44.7	15.6	49.8	2.7	1.0	2.1	30.8	13.0	30.9
	SE05	3.6	0.9	2.3	39.8	7.0	25.4	2.6	0.9	1.5	26.4	5.7	15.5
	Tau03	16.9	20.6	19.7	58.1	221.8	57.8	11.2	14.9	12.7	35.5	140.2	32.3
×	Tau03 opt	3.5	1.0	2.3	12.4	11.0	9.6	2.6	1.6	1.4	7.8	10.2	5.7

SCC-based: Cou99, Cou99 shy-, Cou99 shy, GV04, CVWY90, SE05, Tau03, Tau03 opt

NDFS: Cou99, Cou99 shy-, Cou99 shy, GV04, CVWY90, SE05, Tau03, Tau03 opt

nal algorithm [19]; informally, it visits all the successors of a state even if it could have answered after having visited the first.

Experiments based only on random graphs can be misleading. To emphasize the advantage of TGBAs and SCC-based emptiness checks, we have verified concrete formulæ against concrete models. For this purpose, we have treated one example presented by Geldenhuys and Valmari [8] modeling an algorithm of election in an arbitrary network (this model is also experimented by Schwoon and Esparza [17]). Among the three variations they presented [8], Table 2 collects our results only for the second one, checked against their 9 formulæ (labeled from A to I). Values for the other, less significant variations can be computed using the benchmark scripts distributed with Spot.

Each square corresponds to a given formula. At the top of a square is indicated the label of the formula as well as the product size (in terms of number of states, transitions and acceptance conditions). Moreover, a symbol indicates if the product is empty (∅) or if an accepting run exists (↻). For each algorithm we give the number of distinct states

Table 2. Leader election algorithm in an arbitrary network

		A(287922, 1221437, 1)↻			B(287922, 1222805, 1)↻			C (47887, 134916, 0) ∅		
	Cou99	365	365	365	365	365	365	47887	134916	115
×	Cou99 shy-	365	1356	1358	365	1356	1358	47887	134916	226
×	Cou99 shy	365	1356	1358	365	1356	1358	47887	134916	226
	GV04	365	365	365	365	365	365	47887	134916	115
	CVWY90	17693	91145	902	448	789	787	47887	269831	115
	SE05	17693	90803	564	448	449	449	47887	269831	115
	Tau03	17702	187964	911	448	1876	787			
×	Tau03 opt	365	365	366	365	365	366	47887	134916	115
		D (289812, 1232783, 1) ∅			E (145400, 413351, 0) ↻			F(289812, 1225799, 1)∅		
	Cou99	289812	1232783	145172	365	365	365	289812	1225799	145172
×	Cou99 shy-	289812	1232783	145666	365	706	708	289812	1225799	145666
×	Cou99 shy	289812	1232783	145304	365	706	708	289812	1225799	145304
	GV04	289812	1232783	145172	365	365	365	289812	1225799	145172
	CVWY90	289812	1642497	1145	365	703	704	289812	1635513	1145
	SE05	289812	1642497	1145	365	365	366	289812	1635513	1145
	Tau03	289812	2875280	1145				289812	2861312	1145
×	Tau03 opt	289812	1642497	1145	365	365	366	289812	1635513	1145
		G (241808, 687630, 1) ↻			H(728132, 2080615, 4)↻			I (728132, 2076619, 4) ∅		
	Cou99	557	557	557	145847	413799	145172	728132	2076619	145172
×	Cou99 shy-	557	1087	1089	145847	414229	145303	728132	2076619	145307
×	Cou99 shy	557	1087	1089	145847	414229	145257	728132	2076619	145257
	GV04	557	557	557	145847	413799	145172	728132	2076619	145172
	CVWY90	557	895	896	178543	511930	1388	728132	2489217	1172
	SE05	557	557	558	178543	504468	1145	728132	2489217	1172
	Tau03	566	1249	905	178551	1604336	1454	728132	6631906	1454
×	Tau03 opt	557	557	558	145847	827149	1454	728132	4555287	1454

visited, traversed transitions, and the maximal size of the stack. No measures have been done for Tau03 on TGBAs without acceptance condition.

The complete reachability graph (i.e., without partial order reduction—the conclusion for the reduced graphs are similar, only with smaller figures) of the model has been generated from its Promela specification using Spin [13]. Then the corresponding TGBA has been introduced in Spot and the formulæ translated into TGBAs using also Spot. Though this is not generally the case, on this example the sizes of the degeneralized product and of the generalized one are identical. This is why Cou99 and GV04 perform equally well. The original implementation of Cou99 [3] would have used far less stack, but visited twice as many transitions (for instance on formula F the results for the implementation of Cou99 without rem are $\langle 289812, 2451598, 1145\rangle$).

These runs confirm the conclusions of Schwoon and Esparza [17]. SE05 always performs better than CVWY90 (formulæ A, B, E, G and H); and SCC-based algorithms Cou99 and GV04 perform better than NDFS ones (formulæ A and H).

To conclude our experimentation and focus on multiple acceptance conditions, we present complementary measures for a simple client-server example where c clients communicate with s servers via a duplex channel. Any client can send a request, then

Table 3. Client-server algorithm

		3 cl., 1 serv.			∅	3 cl., 1 serv., fair			∅
	Cou99	a	783	2371	511	b	783	2371	511
×	Cou99 shy-	a	783	2371	710	b	783	2371	710
×	Cou99 shy	a	783	2371	519	b	783	2371	519
	GV04	a	783	2371	511	b'	2005	6627	550
	CVWY90	a	783	2897	237	b'	2005	7771	251
	SE05	a	783	2897	237	b'	2005	7771	251
	Tau03	a	783	5268	238	b	783	10143	264
×	Tau03 opt	a	783	2897	237	b	783	8200	264
		3 cl., 3 serv.			↻	3 cl., 3 serv., fair			∅
	Cou99	c	631	839	159	d	21394	85387	11465
×	Cou99 shy-	c	631	1153	487	d	21394	85387	17133
×	Cou99 shy	c	1170	1914	401	d	21394	85387	11469
	GV04	c	631	839	159	d'	77979	339876	11521
	CVWY90	c	631	1513	159	d'	77979	410877	5632
	SE05	c	631	1499	159	d'	77979	410877	5632
	Tau03	c	899	3373	191	d	21394	415551	5099
×	Tau03 opt	c	631	1499	159	d	21394	331587	5060

sizes of products			
ref.	# st.	# tr.	# cond.
a	783	2371	1
b	783	2371	5
b'	2005	6627	1
c	21394	85387	1
d	21394	85387	7
d'	77979	339876	1

some server will answer that client. The property we check is that if the first client sends a request it will get an answer. This property is only satisfied for 1 client and is otherwise false unless weak fairness is assumed. Table 3 shows the measures. One can indeed see that the property is not satisfied in the case of 3 clients without fairness. The interesting point is that the additional acceptance conditions used for fairness constraints comes at no cost for Cou99 while the cost is high for other algorithms. This is obvious on the cases with 1 client (and can be generalized), however we cannot directly compare the product sizes for 3 clients as the fair case is empty while the unfair case is not.

4 Heuristics and Optimizations

4.1 Heuristics for SCC-Based Algorithms

The two shy variants of Cou99 measured in these tables use the fact that line c15 in Fig. 2 does not enforce any order on the successors. Cou99 will simply use the physical order of the successors in memory, so the $succ$ member of $todo$ items can be efficiently represented as an iterator. The Cou99 shy- variant orders successors to visit those that are already in H first before visiting new states. Doing so sounds natural because it favors merges of SCCs upon pushs. Cou99 shy works similarly, but it considers the successors of the whole top SCC instead of selecting a successor only among the successors of the state at the top of $todo$ (in practice $todo$ is merged like SCC).

Because Cou99 shy- and Cou99 shy have to reorder the successors before executing line c15, the $succ$ field of $todo$ entries cannot be represented as an iterator. To be fair our measures of the stack size of these two variants also account for the number of states of each $succ$ field. Also, while Cou99 makes it possible to compute successors of a state one by one on-the-fly, this is not possible for shy variants who need *all* suc-

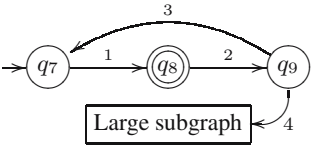


Fig. 4. Problematic case for SE05

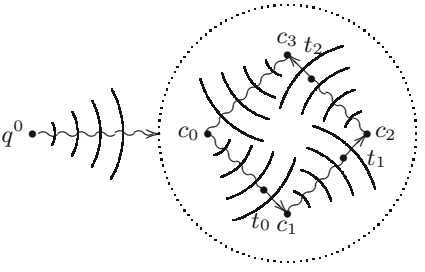


Fig. 5. Computing an accepting run for a TGBA

cessors to reorder them. This difference is apparent in the number of transitions visited: shy variants compute more transitions than plain Cou99.

These heuristics have a controversial effect on performance. Often, they will indeed visit less states, but in counterpart they compute more transitions and require more stack space. On non-empty automata, it is possible to find cases (e.g., bottom left of Table 3) where the variants visit more states. One issue with measuring on-the-fly emptiness checks is that they exit as soon as they can: a more complex algorithm may exit before an efficient one if it luckily picks successors in the right order. (Apart from these two shy variants, all the other algorithms implemented here visit states in the same DFS order; this ensures equitable measurements.) This confirms observations of Geldenhuys and Valmari [8] who tested other heuristics, none of which appeared better either.

4.2 A New Nested DFS Algorithm

Fig. 4 illustrates a case where SE05 could be improved. Arcs are labeled by their depth-first order. SE05 is defined on Büchi automata with accepting states. In its first DFS, if either q_9 or q_7 are accepting, then SE05 can report a violation. If q_8 is accepting, the accepting cycle (q_8, q_9, q_7, q_8) cannot be detected by the first DFS: it will only be found by the second DFS performed *after* the large subgraph have been explored.

The first DFS could detect an accepting cycle when visiting the third arc if it knew whether an accepting state exists between q_7 and q_9. We propose to associate each state q in the DFS stack with the number $W[q]$ of accepting states in the DFS path from q^0 to q. Therefore checking the existence of an accepting state between q_7 and q_9, amounts to testing whether $W[q_9] - W[q_7] > 0$.

This technique can be generalized to multiple acceptance conditions using a vector of counters. We implemented it in Tau03 opt. Its effect can be observed on TGBAs with a single acceptance condition, where SE05 and Tau03 opt differ only on this last optimization. For instance see formulæ A and B in Table 2.

Fig. 6 presents Tau03 opt. This new algorithm uses the technique of Tau03 to handle multiple acceptance conditions, but simplifies its logic and also implements all the optimizations introduced by SE05.

On Table 1 the reason why Tau03 opt outperforms GV04 in terms of visited states is that the latter works on a degeneralized automaton (this is confirmed when comparing Cou99 with Tau03 opt); however the way Tau03 opt nests multiple DFSs to handle multiple acceptance conditions causes more transitions to be visited than GV04.

```
t1  // Let ⟨Σ, Q, δ, q^0, F⟩ be the
t2  // input automaton to check.
t3  H: map of Q ↦ ⟨color ∈ {cyan, blue},
t4                 acc ⊆ F⟩
t5  W: map of Q ↦ →map of F ↦ →ℕ
t6  weight: map of F ↦ →ℕ
t7
t8  main():
t9    forall f ∈ F, weight[f] ← 0
t10   return dfs_blue(q^0)
t11
t12 propagate(s ∈ Q, Acc ⊆ F, t ∈ Q):
t13   ⟨tcol, tacc⟩ ← H[t]
t14   if tcol = cyan ∧ F = (H[s].acc ∪ Acc ∪
t15       tacc ∪ {f ∈ F | weight[f] > W[t][f]})
t16     return ⊥
t17   else if Acc ⊈ tacc
t18     H[t].acc ← tacc ∪ Acc
t19     if dfs_red(t, Acc) = ⊥
t20       return ⊥
t21   return ⊤

t23 dfs_blue(s ∈ Q):
t24   H[s] ← ⟨cyan, ∅⟩
t25   W[s] ← weight
t26   forall ⟨l, a, t⟩ such that ⟨s, l, a, t⟩ ∈ δ
t27     if t ∉ H
t28       forall f ∈ a
t29         weight[f] ← weight[f] + 1
t30       if dfs_blue(t) = ⊥
t31         return ⊥
t32       forall f ∈ a
t33         weight[f] ← weight[f] − 1
t34     if propagate(s, H[s].acc ∪ a, t) = ⊥
t35       return ⊥
t36   H[s].color ← blue
t37   delete W[s]
t38   return ⊤
t39
t40 dfs_red(s ∈ Q, Acc ⊆ F):
t41   forall ⟨l, a, t⟩ such that ⟨s, l, a, t⟩ ∈ δ
t42     if t ∈ H ∧ propagate(s, Acc, t) = ⊥
t43       return ⊥
t44   return ⊤
```

Fig. 6. A variation on the emptiness-check algorithm of Tauriainen [19]

5 Computing Accepting Runs for Generalized Automata

When a product space is found *not* to be empty, it means the system does not verify the formula it is checked against. An important step is to provide the user with a counterexample, showing an actual faulty execution of the system. Such a counterexample is an accepting run of the product automata. It can often be produced as a side-effect of the emptiness check, or afterwards by reusing some data of the check.

In emptiness-check algorithms that work on degeneralized automata, exhibiting an accepting run if one exists is straightforward. In NDFS-based algorithms (CVWY90, SE05) that run is the contents of the stack. For GV04, Geldenhuys and Valmari [9] showed how to use an extra integer per stack state to produce an accepting run.

In this section we present two techniques to extract accepting runs from the data structures of the algorithms that work on generalized automata: Cou99, Tau03, and their variants. Both techniques try to compute a short accepting cycle using successive breadth-first searches (BFSs) and then construct the shortest prefix leading to this cycle.

Accepting runs with Cou99. When Cou99 returns $\bot$ it means an accepting SCC is reachable from q^0. Fig. 5 shows this SCC as a dotted circle. A state s can easily be told to belong to this SCC by checking whether $H[s] \geqslant SCC.\text{top}().root$.

Because the SCC is accepting, from any of its states there exists a circuit labeled by all acceptance conditions. This circuit may cross the same transitions several times.

Table 4. Comparison of algorithms for computing accepting cycles

	Random formulæ				Human-generated formulæ			
Algorithm	formula's cond.		fair		formula's cond.		fair	
$d = 0.001$	*2328 (1318)*		*2188*		*2308 (2127)*		*1951*	
Cou99	1.9	2.0	17.1	12.7	1.3	1.3	12.3	9.0
Cou99 shy-	1.5	1.5	15.7	11.7	1.0	1.0	11.6	8.4
Cou99 shy	1.5	1.5	15.0	11.0	1.0	1.0	11.0	7.9
Tau03	10.9	9.9	237.6	64.7	8.7	9.5	205.9	53.9
Tau03 opt	2.8	6.8	64.5	18.5	2.2	7.4	53.4	16.4
$d = 0.002$	*2716 (1488)*		*2695*		*2569 (2304)*		*2548*	
Cou99	1.3	1.5	15.2	10.6	0.9	1.0	9.7	6.6
Cou99 shy-	0.9	0.9	14.0	9.4	0.6	0.7	8.8	5.9
Cou99 shy	0.9	0.9	13.3	8.6	0.6	0.6	8.5	5.6
Tau03	10.8	8.5	225.1	61.3	7.7	7.1	153.5	40.9
Tau03 opt	2.6	4.8	61.9	17.8	1.8	4.5	43.6	13.9
$d = 0.01$	*2978 (1569)*		*2979*		*2766 (2441)*		*2765*	
Cou99	0.8	1.0	12.5	7.4	0.5	0.6	7.1	4.2
Cou99 shy-	0.4	0.4	9.8	4.9	0.2	0.2	5.5	2.9
Cou99 shy	0.4	0.4	9.2	4.1	0.2	0.2	5.1	2.5
Tau03	18.1	16.9	210.0	58.1	12.6	11.2	139.4	35.5
Tau03 opt	1.7	3.5	43.0	12.4	1.1	2.6	25.6	7.8

Therefore, it is easier to construct an accepting cycle as a series of independent parts that can each visit a transition at most once, and that each brings new acceptance conditions.

The algorithm thus works as follows. Let $\mathcal{F}_0$ be the set of all acceptance conditions. From a state c_0 of the SCC, start a BFS (restricted to the SCC) to construct a path to the closest transition t_0 that has some acceptance conditions F_0 so that $F_0 \cap \mathcal{F}_0 \not= \emptyset$. Let $\mathcal{F}_1 = \mathcal{F}_0 \setminus F_0$ be the set of remaining acceptance conditions. Repeat the BFS from c_1 (the output of t_0) until a transition t_1 is found with acceptance conditions F_1 that intersect $\mathcal{F}_1$. Iterate until $\mathcal{F}_n = \emptyset$. Finally use a last BFS to compute the shortest path from c_n back to c_0, closing the cycle. This algorithm was presented by Latvala and Heljanko [15] using the root of the SCC as c_0. However the choice of c_0 can be arbitrary because we are in a SCC. Since we know that the transition that caused Cou99 to exit (the one corresponding to the last execution of line c15) is necessarily part of the acceptance cycle, it seems wiser to use either its source or its destination as c_0.

As far as the prefix is concerned, a list of states from q^0 to c_0 can be easily constructed while unwinding the *todo* stack. However this prefix may not be the shortest possible prefix leading to the accepting cycle, so a similar idea would be to use a BFS to construct the shortest path between q^0 and any state of the cycle, this path can be constrained to visit the SCCs in increasing order to limit the scope of the BFS.

Accepting runs with Tau03. Computing accepting runs for generalized NDFS algorithms such as Tau03 or Tau03 opt is more embarrassing, because the resulting data do not provide structural information as useful as a SCC that would restrict our search.We know that the last s for which line t34 was executed belongs to an accepting cycle. From

this state $c_0 = s$ we first perform a nested DFS to collect a set of transitions $\mathcal{T}$ that (1) are each on a cycle back to c_0, and (2) will, together, cover $\mathcal{F}$.

This collection of cycles could be used to construct an accepting cycle, but since we are trying to create short runs we decided to connect these collected transitions directly. Therefore we perform a BFS to compute the shortest path from c_0 to a transition t_0 of $\mathcal{T}$, and from there another BFS to find the shortest path to another t_1 of $\mathcal{T}$, etc. Closing the cycle and computing the prefix can be done like for SCC-based algorithms.

Table 4 uses the layout of Table 1. For each setup, the two values are the number of states visited to construct the cycle part of the accepting run, and the size of the search space for this cycle. They are expressed as a percentage of the number of states of the input TGBA. For Cou99 the cycle's search space is the top SCC, and for Tau03 the search space contains all states in H. (A state is counted as many times as it is visited.)

As the table shows, the absence of structural information in Tau03 makes the computation more costly, since the search space is larger. For Cou99, the search is contained in a small subgraph (the top SCC), which justifies the use of BFSs. The "fair" columns show that with more acceptance conditions in the system the algorithms need to traverse the search space more times. Surprisingly, the size of the search space for Cou99 shy and Cou99 shy- is smaller than that of Cou99; this is counter-intuitive because our heuristics aim at favoring merges of SCCs.

During our experiments we observed that the size of accepting runs produced by such BFS-based algorithms were significantly smaller than those obtained directly from the stack of NDFS algorithms. A deeper study of existing algorithms, weighting minimization against computational complexity still has to be done (Gastin et al. [7] provide some initial clues).

6 Conclusion

In this paper we have stressed the importance of dealing with *generalized* Büchi automata in emptiness-check algorithms. Our experiments on existing algorithms showed that SCC-based ones clearly outrank NDFSs; this completes the results of Schwoon and Esparza [17], who studied emptiness checks of standard Büchi automata.

Although we have not implemented it, the generalized algorithms presented here can be used in conjunction with the bit-state hashing technique [13, p. 206] if done carefully. The bit-state hashing should not be applied to states that belong to the first-level DFS: those states need to be perfectly hashed. The application to Tau03 is discussed by Tauriainen [20]. In SCC-based algorithms the restriction extends to all states that belong to SCCs in the SCC stack. In other words, bit-state hashing can only be applied to states from removed MSCCs; this limits its usefulness.

To give NDFS-based algorithms a chance to compete with SCC-based ones, we introduced (1) a new optimization to detect some accepting cycles earlier, and (2) a new algorithm (Tau03 opt) that mixes all the optimizations of SE05 with the multiple acceptance condition capability of Tau03. Although Tau03 opt surpasses other NDFS algorithms, our experiments still show that SCC-based algorithms perform better.

To complete our TGBA verification framework, we finally introduced algorithms to extract accepting runs. Here again, our results are in favor of Cou99.

All the algorithms presented and measured here are implemented in our model-checking library, Spot [4], available at `http://spot.lip6.fr`. The distribution of Spot includes the scripts we used for our experiments. They can be adjusted to different configurations, and can output more statistics than we could present in these pages. For instance they also verify the reduced state spaces generated from the examples using Spin's partial-order reduction [13, p. 192].

References

[1] I. Černá and R. Pelánek. Relating hierarchy of temporal properties to model checking. In *Proc. of MFCS'03*, volume 2747 of *LNCS*, pages 318–327. Springer-Verlag, Aug. 2003.

[2] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithm for the verification of temporal properties. In *Proc. of CAV'90*, volume 531 of *LNCS*, pages 233–242. Springer-Verlag, 1991.

[3] J.-M. Couvreur. On-the-fly verification of temporal logic. In *Proc. of FM'99*, volume 1708 of *LNCS*, pages 253–271. Springer-Verlag, Sept. 1999.

[4] A. Duret-Lutz and D. Poitrenaud. Spot: an extensible model checking library using transition-based generalized Büchi automata. In *Proc. of MASCOTS'04*, pages 76–83. IEEE Computer Society, Oct. 2004.

[5] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property specification patterns for finite-state verification. In *Proc. of FMSP'98*, pages 7–15. ACM, Mar. 1998.

[6] K. Etessami and G. J. Holzmann. Optimizing Büchi automata. In *Proc. of Concur'00*, volume 1877 of *LNCS*, pages 153–167. Springer-Verlag, 2000.

[7] P. Gastin, P. Moro, and M. Zeitoun. Minimization of counterexamples in SPIN. In *Proc. of SPIN'04*, volume 2989 of *LNCS*, pages 92–108. Springer-Verlag, 2004.

[8] J. Geldenhuys and A. Valmari. Tarjan's algorithm makes on-the-fly LTL verification more efficient. In *Proc. of TACAS'04*, volume 2988 of *LNCS*, pages 205–219. Springer-Verlag, 2004.

[9] J. Geldenhuys and A. Valmari. More efficient on-the-fly LTL verification with Tarjan's algorithm. *Theoretical Computer Science*, 2005. To appear: conference paper selected for journal publication.

[10] D. Giannakopoulou and F. Lerda. From states to transitions: Improving translation of LTL formulæ to Büchi automata. In *Proc. of FORTE'02*, volume 2529 of *LNCS*, pages 308–326. Springer-Verlag, Nov. 2002.

[11] P. Godefroid and G. J. Holzmann. On the verification of temporal properties. In *Proc. of PSTV'93*, volume C-16 of *IFIP Transactions*, pages 109–124. North-Holland, May 1993.

[12] M. Hammer, A. Knapp, and S. Merz. Truly on-the-fly LTL model checking. In *Proc. of TACAS'05*, LNCS. Springer-Verlag, Apr. 2005.

[13] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003. ISBN 0-321-22862-6.

[14] G. J. Holzmann, D. A. Peled, and M. Yannakakis. On nested depth first search. In *Proc. of SPIN'96*, volume 32 of *DIMACS*. AMS, May 1996.

[15] T. Latvala and K. Heljanko. Coping with strong fairness. *Fundamenta Informaticae*, 43 (1–4):1–19, 2000.

[16] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. of POPL'85*, pages 97–107. ACM, 1985.

[17] S. Schwoon and J. Esparza. A note on on-the-fly verification algorithms. In *Proc. of TACAS'05*, LNCS. Springer-Verlag, Apr. 2005. To appear.

[18] F. Somenzi and R. Bloem. Efficient Büchi automata for LTL formulæ. In *Proc. of CAV'00*, volume 1855 of *LNCS*, pages 247–263. Springer-Verlag, 2000.

[19] H. Tauriainen. On translating linear temporal logic into alternating and nondeterministic automata. Research Report A83, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, Dec. 2003.

[20] H. Tauriainen. Nested emptiness search for generalized Büchi automata. In *Proc. of ACSD'04*, pages 165–174. IEEE Computer Society, June 2004.

[21] H. Tauriainen. A randomized testbench for algorithms translating linear temporal logic formulæ into Büchi automata. In *Proc. of CS&P'99*, pages 251–262, Sept. 1999.

[22] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Proc. of Banff'94*, volume 1043 of *LNCS*, pages 238–266. Springer-Verlag, 1996.

Stuttering Congruence for χ

Bas Luttik[1,2] and Nikola Trčka[1,⋆]

[1] Department of Mathematics and Computer Science, Eindhoven University of Technology, P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands
[2] CWI, P.O. Box 94079, NL-1090 GB Amsterdam, The Netherlands

Abstract. The language χ is a modeling and simulation language which is currently mainly used to analyse and optimize the performance of industrial systems. To be able to also verify functional properties of a system using a χ model, part of the language has been given a formal semantics. Rather than implementing a new model checker for χ, the philosophy is to provide automatic translations from χ into the specification languages of existing state-of-the-art model checkers such as, e.g., SPIN and UPPAAL.

In this paper, we propose for χ a notion of stuttering congruence, which is an adaptation of the notion of stuttering equivalence. We prove that our notion preserves the validity of $\text{CTL}^*_{-\text{X}}$ formulas, that it preserves deadlock, and that it is indeed a congruence with respect to the constructs of χ. We also indicate how our notion is to be used to establish confidence in the correctness of a translation from χ into PROMELA.

1 Introduction

The language χ [19] is a modeling language developed for detecting design flaws and for optimizing performance of industrial systems (machines, manufacturing lines, warehouses, factories, etc.) by simulation. Quite a few case studies have shown the usefulness of χ in an industrial context [12,6,10,21]; simulation turns out to be a powerful technique for doing performance analysis such as approximating throughput and cycle time. However, for the verification of functional properties such as, e.g., deadlock freedom, simulation is less suitable. To be able to also do verification with χ, either verification tools have to be developed especially for χ, or existing verification tools and techniques have to be made available for use with χ. Currently, the latter approach is pursued.

The idea is to extend χ with facilities for doing formal verification by establishing a connection with other verification tools on the level of the specification language. That is, formal verification of a χ model is done by first translating it into the input language of some model checker and then performing the actual verification. Preferably, the translation closely resembles the original, so that a counterexamples produced by the model checker can be related to the original specification. The suitability of this approach was shown in [2,3], where a

⋆ Research supported by the Netherlands Organization for Scientific Research (NWO) under project number 612.064.205.

P. Godefroid (Ed.): SPIN 2005, LNCS 3639, pp. 185–199, 2005.

χ model of a turntable machine was translated to PROMELA [14], μCRL [1] and UPPAAL timed automata [15], and then verified in SPIN, CADP [9] and UPPAAL, respectively.

In [20], the translation of χ specifications into PROMELA is discussed in more generality. The translation proceeds in two phases. The first phase, which we call the *preprocessing phase*, consists of a transformation of the χ model in an attempt to eliminate all constructs that do not directly map to PROMELA constructs. For instance, χ has an explicit construct for parallel composition which facilitates nested parallelism, whereas PROMELA only allows the (implicit) parallel composition of sequential PROMELA processes; so in the preprocessing phase the nested parallelism in the χ model is eliminated. If the result after the preprocessing phase is a χ model that only has constructions with a direct translation into PROMELA, then it can be translated to a PROMELA model; this phase is called the *translation phase*.

The main difficulty for establishing the correctness of the whole translation is that usually the two languages do not have a formal semantics in common. An advantage of the two-phase approach sketched above then is that the preprocessing phase of the translation, which is usually the most involved part, takes place entirely within the realm of χ. Therefore, a correctness proof for this phase only involves the formal semantics of χ. An additional advantage of the two-phase approach is that the preprocessing phase (and its correctness proof) is potentially reusable, e.g., when defining a translation from χ to some other language.

The appropriate correctness criterion for a translation depends of course on the application. If the purpose is to establish that a χ model is deadlock-free, then the translation should preserve deadlock. If the purpose is to do LTL/CTL model checking, then the translation should preserve the validity of LTL/CTL formulas. In all cases, establishing the desired preservation of properties directly is usually cumbersome. It is often more convenient to relate the χ model and its transformation by establishing that they are related according to some behavioral equivalence pertaining to the operational semantics of χ. The purpose of this paper is to define a behavioral equivalence that can be used to establish the correctness of the preprocessing phase of translations of χ models into the language of state-based model checkers such as, e.g., SPIN or UPPAAL.

Of course, such an equivalence should then preserve the relevant properties; in our case it will preserve deadlock and the validity of (state-based) $\mathrm{CTL}^*_{-\mathrm{X}}$ formulas. (We do not require that the validity of formulas containing the next time operator X be preserved, in order to achieve sufficient flexibility for translations.) Its intended application for establishing the correctness of syntactic translations puts some further requirements on the notion. For instance, since the transformations are generally defined in a compositional manner, it is particularly convenient if the equivalence is a congruence with respect to the syntactic constructs of χ. Also, it is desirable that it is defined on the operational semantics of χ as directly as possible, i.e., it should be bisimulation-like. We propose a notion of stuttering congruence that meets these requirements. We present it in

the context of the language χ, but since the constructs of χ are fairly standard, we think that our notion can be of use for other languages too.

The paper is organized as follows. In Section 2 we present the syntax and the operational semantics of the discrete-event and untimed part of the language χ. We use the operational semantics to define when a χ-process has a deadlock, and we give the semantics of $\text{CTL}^*_{-\text{X}}$ formulas with respect to χ processes. In Section 3, we propose an adaptation of divergence blind stuttering bisimilarity [18]. We add to it a termination condition, which takes care of the distinction between successful and unsuccessful termination present in χ, and a divergence condition, which is needed both for the preservation of deadlock and preservation of $\text{CTL}^*_{-\text{X}}$. We prove that our version of stuttering bisimilarity is an equivalence relation and that it indeed preserves deadlock and the validity of (state-based) $\text{CTL}^*_{-\text{X}}$ formulas. In Section 4 we argue that stuttering bisimilarity as defined in Section 3 is not a congruence. So we adapt it further by excluding send and receive transitions as stuttering steps and by adding a root condition. The resulting notion we call stuttering congruence and we prove that it is indeed a congruence with respect to the syntactic constructs of the discrete-event, untimed part of χ. In Section 5 we briefly indicate how our notion of stuttering congruence can be used to establish part of the correctness of the translation proposed in [20]. The paper ends with a conclusion. For detailed proofs of the results in this paper we refer to the full version [16].

2 The language χ

In this section we present the syntax and operational semantics of χ. We also define the notion of deadlock and the semantics of $\text{CTL}^*_{-\text{X}}$ for χ processes. We use the formalization of χ proposed in [4], but without the time support and with a few minor differences that we shall mention on the fly.

2.1 Syntax and semantics

There are several predefined data types in χ, but they are not relevant for the present paper. For our purposes, it is enough to presuppose a set of variables V, a set of data values D, a set of data expressions E that includes D and V, and a set of boolean expressions B that includes the set of truth values $\{\mathit{true}, \mathit{false}\}$.

Definition 1. *A partial mapping $\sigma : V \rightharpoonup D$ with a finite domain (denoted $\text{dom}(\sigma)$) is called a* state. *The set of all states is denoted Σ.*

To correctly override global variables by local ones of the same name, we use the function $\gamma : \Sigma \times \Sigma \to \Sigma$ defined as:

$$\begin{aligned} &\text{dom}(\gamma(\sigma_1, \sigma_2)) = \text{dom}(\sigma_1) \cup \text{dom}(\sigma_2) \\ &\gamma(\sigma_1, \sigma_2)(x) = \begin{cases} \sigma_1(x), & \text{if } x \in \text{dom}(\sigma_1) \\ \sigma_2(x), & \text{if } x \in \text{dom}(\sigma_2) \backslash \text{dom}(\sigma_1). \end{cases} \end{aligned}$$

We assume that σ also extends to data expressions ($\sigma : E \rightharpoonup D$) and to boolean expressions ($\sigma : B \to \{\mathit{true}, \mathit{false}\}$). In the latter case we require σ to be total.

We now give the syntax of χ. The set of atomic processes A, and the set of all χ process terms P, are generated by the following grammar:

$$\begin{aligned} a &::= \varepsilon \mid \delta \mid \mathit{skip} \mid x := e \mid m!e \mid m?x \\ p &::= a \mid b :\to p \mid p\,;p \mid p \,[\!]\, p \mid p^* \mid p \parallel p \mid [\![s \mid p]\!] \mid \partial(p) \ . \end{aligned}$$

Here $a \in A$, $p \in P$, $x \in V$, $e \in E$, $b \in B$, $s \in \Sigma$ and $m \in M$, where M is a set of channel names.

Elements of the set $C = P \times \Sigma$ we call *configurations*. They represent processes together with their context. If $c = \langle p, \sigma \rangle$, then σ is the *state* of c. The semantics of χ is given in terms of configurations.

We make a distinction between successful and unsuccessful termination. The statement $c{\downarrow}$ denotes that $c \in C$ successfully terminates. The statement $c \xrightarrow{a} c'$ denotes that $c \in C$ can execute the action a and transform into the configuration c'. The set of actions that can be performed (denoted $\mathcal{A}$) consists of the internal action τ, the assignment action $aa(x, d)$, the send action $sa(m, d)$, the receive action $ra(m, d)$ and the communication action $ca(m, d)$, where $x \in V$, $m \in M$ and $d \in D$.

Atomic processes We explain each atomic process informally; the operational rules are given in Table 1.

The constant δ stands for the deadlock process. It cannot execute an action nor terminate successfully. The empty process ε cannot do an action either, but it is considered successfully terminated. The *skip* process performs the internal action τ (and terminates successfully). The assignment process $x := e$ assigns to x the value of the expression e according to the current state. The send process $m!e$ outputs the value of e (in the current state) along channel m. The receive process $m?x$ inputs a value along channel m and assigns it to x.

Table 1. Operational semantics for atomic processes

$$\frac{}{\langle \varepsilon, \sigma \rangle{\downarrow}}\ 1 \qquad \frac{}{\langle \mathit{skip}, \sigma \rangle \xrightarrow{\tau} \langle \varepsilon, \sigma \rangle}\ 2 \qquad \frac{\sigma(e) = d}{\langle x := e, \sigma \rangle \xrightarrow{aa(x,d)} \langle \varepsilon, \gamma(\{x \shortmid\!\to d\}, \sigma) \rangle}\ 3$$

$$\frac{\sigma(e) = d}{\langle m!e, \sigma \rangle \xrightarrow{sa(m,d)} \langle \varepsilon, \sigma \rangle}\ 4 \qquad \frac{}{\langle m?x, \sigma \rangle \xrightarrow{ra(m,d)} \langle \varepsilon, \gamma(\{x \shortmid\!\to d\}, \sigma) \rangle}\ 5$$

Compound processes Here we give an informal explanation for each of the seven operators; the operational rules are given in Table 2.

The guarded process $b :\to p$ behaves as p when the value of the guard $b \in B$ is *true* (in the current state). The sequential composition $p\,;q$ behaves as p

followed by the process q. The alternative composition $p \,[\!]\, q$ stands for a non-deterministic choice between p and q. The process p^* behaves as p, executed zero (successful termination), or more times. The parallel composition operator $\|$ executes p and q concurrently in an interleaved fashion. In addition, if one of the processes can execute a send action and the other one can execute a receive action on the same channel, then they can also communicate, i.e. $p \,\|\, q$ can also execute the communication action on this channel. The scope operator is used for declarations of local variables. The process $[\![s \mid p]\!]$ behaves as p in a local state s. In contrast to [4] and [19], channel declarations are not allowed, i.e. channels are global. Finally, the encapsulation operator ∂ disables all send and receive actions of a process. This is slightly more restrictive than in [4] and [19] where ∂ is parameterized by a set of actions that should be blocked, but corresponds to current practice.

Table 2. Operational semantics for composed processes

$$\frac{\sigma(b) = \mathit{true},\ \langle p,\sigma\rangle\downarrow}{\langle b :\rightarrow p,\sigma\rangle\downarrow}\ 6 \qquad \frac{\sigma(b) = \mathit{true},\ \langle p,\sigma\rangle \xrightarrow{a} \langle p',\sigma'\rangle}{\langle b :\rightarrow p,\sigma\rangle \xrightarrow{a} \langle p',\sigma'\rangle}\ 7$$

$$\frac{\langle p,\sigma\rangle\downarrow,\ \langle q,\sigma\rangle\downarrow}{\langle p\,;q,\sigma\rangle\downarrow}\ 8 \qquad \frac{\langle p,\sigma\rangle\downarrow,\ \langle q,\sigma\rangle \xrightarrow{a} \langle q',\sigma'\rangle}{\langle p\,;q,\sigma\rangle \xrightarrow{a} \langle q',\sigma'\rangle}\ 9 \qquad \frac{\langle p,\sigma\rangle \xrightarrow{a} \langle p',\sigma'\rangle}{\langle p\,;q,\sigma\rangle \xrightarrow{a} \langle p'\,;q,\sigma'\rangle}\ 10$$

$$\frac{\langle p,\sigma\rangle\downarrow}{\langle p \,[\!]\, q,\sigma\rangle\downarrow,\ \langle q \,[\!]\, p,\sigma\rangle\downarrow}\ 11 \qquad \frac{\langle p,\sigma\rangle \xrightarrow{a} \langle p',\sigma'\rangle}{\langle p \,[\!]\, q,\sigma\rangle \xrightarrow{a} \langle p',\sigma'\rangle,\ \langle q \,[\!]\, p,\sigma\rangle \xrightarrow{a} \langle p',\sigma'\rangle}\ 12$$

$$\frac{}{\langle p^*,\sigma\rangle\downarrow}\ 13 \qquad \frac{\langle p,\sigma\rangle \xrightarrow{a} \langle p',\sigma'\rangle}{\langle p^*,\sigma\rangle \xrightarrow{a} \langle p'\,;p^*,\sigma'\rangle}\ 14$$

$$\frac{\langle p,\sigma\rangle\downarrow,\ \langle q,\sigma\rangle\downarrow}{\langle p \,\|\, q,\sigma\rangle\downarrow,\ \langle q \,\|\, p,\sigma\rangle\downarrow}\ 15 \qquad \frac{\langle p,\sigma\rangle \xrightarrow{a} \langle p',\sigma'\rangle}{\langle p \,\|\, q,\sigma\rangle \xrightarrow{a} \langle p' \,\|\, q,\sigma'\rangle,\ \langle q \,\|\, p,\sigma\rangle \xrightarrow{a} \langle q \,\|\, p',\sigma'\rangle}\ 16$$

$$\frac{\langle p,\sigma\rangle \xrightarrow{sa(m,d)} \langle p',\sigma\rangle,\ \langle q,\sigma\rangle \xrightarrow{ra(m,d)} \langle q',\sigma'\rangle}{\langle p \,\|\, q,\sigma\rangle \xrightarrow{ca(m,d)} \langle p' \,\|\, q',\sigma'\rangle,\ \langle q \,\|\, p,\sigma\rangle \xrightarrow{ca(m,d)} \langle q' \,\|\, p',\sigma'\rangle}\ 17$$

$$\frac{\langle p,\gamma(s,\sigma)\rangle\downarrow}{\langle [\![s \mid p]\!],\sigma\rangle\downarrow}\ 18 \qquad \frac{\langle p,\gamma(s,\sigma)\rangle \xrightarrow{a} \langle p',\sigma'\rangle}{\langle [\![s \mid p]\!],\sigma\rangle \xrightarrow{a} \langle [\![\sigma' \upharpoonright \mathrm{dom}(s) \mid p']\!],\gamma(\sigma,\sigma' \upharpoonright \mathrm{dom}(\sigma)\backslash\mathrm{dom}(s))\rangle,}\ 19$$

$$\frac{\langle p,\sigma\rangle\downarrow}{\langle \partial(p),\sigma\rangle\downarrow}\ 20 \qquad \frac{\langle p,\sigma\rangle \xrightarrow{a} \langle p',\sigma'\rangle,\ a \notin \{sa(m,d), ra(m,d)\}}{\langle \partial(p),\sigma\rangle \xrightarrow{a} \langle \partial(p'),\sigma'\rangle}\ 21$$

2.2 Deadlock and $\mathbf{CTL}^*_{-\mathbf{x}}$ in χ

First we give some abbreviations: $c \rightarrow c'$ denotes that there exists $a \in A$ such that $c \xrightarrow{a} c'$; $c \not\rightarrow$denotes that there does not exist a c' such that $c \rightarrow c'$.

Now we define when a configuration has a deadlock.

Definition 2. *A configuration c has a* deadlock *iff there exist $c_0, \ldots, c_n \in C$ such that*

$$c_0 = c,\ c_0 \to \cdots \to c_n,\ c_n \not\to \text{and } c_n \not\downarrow$$

Next, we recall the formulas of the logic $\text{CTL}^*_{-\text{X}}$ [7] and give their semantics. Let AP be a set that we call the set of atomic propositions.

Definition 3. *The formulas of the logic $\text{CTL}^*_{-\text{X}}$ are defined as follows:*

1. *every atomic proposition is a state formula;*
2. *if φ is a state formula, then $\neg\varphi$ is a state formula;*
3. *if φ_1 and φ_2 are state formulas, then $\varphi_1 \wedge \varphi_2$ is a state formula;*
4. *if ψ is a path formula, then $\exists\psi$ is a state formula;*
5. *if φ is a state formula, then φ is a path formula;*
6. *if ψ is a path formula, then $\neg\psi$ is a path formula;*
7. *if ψ_1 and ψ_2 are path formulas, then $\psi_1 \wedge \psi_2$ is a path formula;*
8. *if ψ_1 and ψ_2 are path formulas, then $\psi_1 \mathcal{U} \psi_2$ is a path formula.*

For the satisfaction relation we need the notion of a path.

Definition 4. *A* path *(from a configuration c_0) is an infinite sequence of configurations $c_0, c_1, c_2, \ldots$ such that either:*

- $c_0 \to c_1 \to c_2 \to \cdots$ *or*
- $c_0 \to \cdots \to c_n$, $c_n \not\to$ *and $c_{i+1} = c_i$ for all $i \geq n$.*

If π is a path $c_0, c_1, c_2, \ldots$ then π^i denotes the path $c_i, c_{i+1}, c_{i+2}, \ldots$.

We require that two configurations with the same state satisfy the same atomic propositions by assuming that for each state $\sigma \in \Sigma$ there is a mapping $val_\sigma : AP \to \{true, false\}$.

Definition 5. *We simultaneously define the satisfaction of a state formula φ by a configuration c (notation: $c \vDash \varphi$) and the satisfaction of a path formula ψ by a path π (notation: $\pi \vDash \psi$) as follows:*

1. *$c \vDash \alpha \in AP$ iff $val_c(\alpha) = true$*
2. *$c \vDash \neg\varphi$ iff $c \nvDash \varphi$,*
3. *$c \vDash \varphi_1 \wedge \varphi_2$ iff $c \vDash \varphi_1$ and $c \vDash \varphi_2$,*
4. *$\pi \vDash \varphi$ iff $c \vDash \varphi$ where c is the first configuration in path π,*
5. *$c \vDash \exists\psi$ iff there is a path π from c such that $\pi \vDash \psi$,*
6. *$\pi \vDash \neg\psi$ iff $\pi \nvDash \psi$,*
7. *$\pi \vDash \psi_1 \wedge \psi_2$ iff $\pi \vDash \psi_1$ and $\pi \vDash \psi_2$,*
8. *$\pi \vDash \psi_1 \mathcal{U} \psi_2$ iff there exists $j \geq 0$ such that $\pi^j \vDash \psi_2$ and $\pi^i \vDash \psi_1$ for all $i < j$.*

3 Stuttering Bisimilarity

Stuttering equivalence was originally proposed and proved to preserve the validity of $\text{CTL}^*_{-\text{X}}$ formulas by Browne, Clarke and Grumberg [5]. They define the notion on maximal paths associated with total Kripke structures, i.e., Kripke structures without deadlocked states. De Nicola and Vaandrager [18] drop the requirement that Kripke structures are total, and provide a definition of stuttering equivalence that proceeds via a notion of divergence blind stuttering bisimilarity defined on the Kripke structures themselves. Groote and Vaandrager [13] give an efficient algorithm for deciding this equivalence. For alternative definitions of stuttering equivalence see also [8,17].

We take divergence blind stuttering bisimilarity of [18] as a starting point, and add to it two conditions:

1. a *termination condition* that ensures a proper handling of the distinction between successful and unsuccessful termination as it is present in χ; and
2. a *divergence condition* similar to one that appears in [11] to ensure the preservation of deadlock and the preservation of $\text{CTL}^*_{-\text{X}}$.

Remark 1. To obtain a notion that coincides with the notion of [5], instead of adding a divergence condition, de Nicola and Vaandrager define a divergence sensitive version of stuttering bisimilarity by extending Kripke structures with a fresh state that serves as a sink-state for deadlocked or divergent states. This approach is not suitable in our case, because it identifies deadlock and livelock, and because it is in conflict with our requirement, mentioned in the introduction, that the equivalence is defined directly on the operational semantics of χ.

Definition 6. *A symmetric relation $R \subseteq C \times C$ is a* stuttering bisimulation *iff, for all $(c, d) \in R$, c and d have the same state and:*

1. *if $c\!\downarrow$, then there exist $d_0, \ldots, d_n \in C$ such that*

$$d_0 = d,\ d_0 \to \cdots \to d_n,\ d_n\!\downarrow \text{ and } cRd_i \text{ for all } i \leq n,$$

2. *if $c \to c'$ for some $c' \in C$, then there exist $d_0, \ldots, d_n \in C$ such that*

$$d_0 = d,\ d_0 \to \cdots \to d_n,\ cRd_i \text{ for all } i \leq n-1,\ \text{and } c'Rd_n,$$

3. *if there exists an infinite sequence $c_0, c_1, c_2, \ldots \in C$ such that*

$$c_0 = c,\ c_0 \to c_1 \to c_2 \to \cdots \text{ and } c_iRd \text{ for all } i \geq 0,$$

then there exist $d' \in C$ and $j > 0$ such that

$$d \to d' \text{ and } c_jRd'.$$

We refer to condition 1 as the termination condition*, to condition 2 as the* transfer condition*, and to condition 3 as the* divergence condition.

A non-empty and finite sequence of configurations we call a *block*. If $B = c_0, \ldots, c_n$ and $C = d_0, \ldots, d_m$ are blocks and R is a stuttering bisimulation, we write BRC when $c_0 R d_0$, $c_n R d_m$ and when, for all $i < n, j < m$, $c_i R d_j$ implies $c_{i+1} R d_j$ or $c_i R d_{j+1}$. Note that, BRC implies CRB.

Definition 7. *Let R be a stuttering bisimulation. Two sequences of configurations Ω_1 and Ω_2 are* R-corresponding *if:*

- *they are both finite and can be partitioned as $\Omega_1 = B^0, \ldots, B^K$ and $\Omega_2 = C^0, \ldots, C^K$ where $B^k R C^k$ for all $0 \leq k \leq K$; or*
- *they are both infinite and can be partitioned as $\Omega_1 = B^0, B^1, B^2 \ldots$ and $\Omega_2 = C^0, C^1, C^2, \ldots$ where $B^k R C^k$ for all $k \geq 0$.*

It is clear that R-correspondence is a symmetric relation. Also note that, if $c_0, \ldots, c_n$ and $d_0, \ldots, d_m$ (similarly for the infinite case) are R-corresponding then for all $i \leq n$ there exists $j \leq m$ and for all $j \leq m$ there exists $i \leq n$ such that $c_i R d_j$.

We now present some properties of a stuttering bisimulation R.

Lemma 1. *If cRd, then for every sequence of configurations $c_0, \ldots, c_n$ such that $c_0 = c$ and $c_0 \to \cdots \to c_n$ there exists an R-corresponding sequence $d_0, \ldots, d_m$ such that $d_0 = d$ and $d_0 \to \cdots \to d_m$.*

Proof. By induction on n. □

Lemma 2. *If cRd, then*

a. for every sequence of configurations $c_0, \ldots, c_n$ such that $c_0 = c$, $c_0 \to \cdots \to c_n$ and $c_n{\downarrow}$ there exists an R-corresponding sequence $d_0, \ldots, d_m$ such that

$$d_0 = d,\ d_0 \to \cdots \to d_m \text{ and } d_m{\downarrow};\quad \text{and}$$

b. for every sequence of configurations $c_0, \ldots, c_n$ such that $c_0 = c$, $c_0 \to \cdots \to c_n$ and $c_n \not\to$ there exists an R-corresponding sequence $d_0, \ldots, d_m$ such that

$$d_0 = d,\ d_0 \to \cdots \to d_m,\ \text{and } d_m \not\to$$

Proof. For both cases, use Lemma 1 to obtain a sequence $d_0, \ldots, d_l$ that R-corresponds to $c_0, \ldots, c_n$ and then extend it to $d_0, \ldots, d_l, \ldots, d_m$ by using Definition 6. (Use the termination condition for the first case, and the transfer and the divergence condition for the second case.) □

Lemma 3. *If cRd, then for every infinite sequence of configurations $c_0, c_1, c_2, \ldots$ such that $c_0 = c$ and $c_0 \to c_1 \to c_2 \to \cdots$, there exists an R-corresponding sequence of configurations $d_0, d_1, d_2, \ldots$ such that $d_0 = d$, $d_0 \to d_1 \to d_2 \to \cdots$.*

Proof. Construct infinite sequences of blocks $C^0, C^1, C^2, \ldots$ and $D^0, D^1, D^2, \ldots$ such that $C^k R D^k$ for all $k \geq 0$, with $C^0, C^1, C^2, \ldots$ a partitioning of $c_0, c_1, c_2, \ldots$ and $D^0, D^1, D^2, \ldots$ a partitioning of $d_0, d_1, d_2, \ldots$ such that $d_0 = d$ and $d_0 \to d_1 \to d_2 \to \cdots$. The construction is by induction on k, where each step delivers the blocks C^k and D^k and the first configurations of C^{k+1} and D^{k+1}. □

Definition 8. *Two configurations c and d are* stuttering bisimilar*, denoted $c \sim_{st} d$, if there exists a stuttering bisimulation R such that cRd.*

We now prove that stuttering bisimilarity is an equivalence relation. The usual way to prove that a bisimulation-like equivalence $\sim$ is transitive, is to suppose that $c \sim d$ and $d \sim e$ are witnessed by bisimulation relations R_1 and R_2 respectively, and then show that $R_1 \circ R_2$ is again a bisimulation relation. However, this method fails here, due to the nature of the divergence condition. We prove transitivity by showing that the transitive closure of a stuttering bisimulation is a stuttering bisimulation.

Lemma 4. *If R is a stuttering bisimulation, then so is $R^+ = \bigcup_{n \in \omega} R^n$.*

Proof. The transitive closure of any symmetric relation is symmetric so R^+ is symmetric. Prove first that for all n and all $(c, d) \in R^n$, c and d have the same state, (c, d) satisfies the termination and transfer condition and if $c_0 = c,\ c_0 \to c_1 \to c_2 \to \cdots$ then there exist $d_0, d_1, d_2, \ldots \in C$ such that $d_0 = d,\ d_0 \to d_1 \to d_2 \to \cdots$ and for all $j \geq 0$ there is $i \geq 0$ such that $c_i\ R^n\ d_j$. Since $R^+ = \bigcup_{n \in \omega} R^n$, it now follows immediately that R^+ satisfies the termination and transfer condition and that, for all $(c, d) \in R^+$, c and d have the same state. Prove that R^+ also satisfies the divergence condition by using that R^+ is symmetric and transitive. □

Theorem 1. *Stuttering bisimilarity on configurations is an equivalence relation.*

Proof. The set $\{(c, c) \mid c \in C\}$ is clearly a stuttering bisimulation, so $\sim_{st}$ is reflexive. Furthermore, that $\sim_{st}$ is symmetric follows directly from the required symmetry of a stuttering bisimulation. It remains to prove transitivity.

Suppose $c \sim_{st} d$ and $d \sim_{st} e$. Then, there exist stuttering bisimulations R_1 and R_2 such that $c R_1 d$ and $d R_2 e$. Let $R = R_1 \cup R_2$. It is not hard to show that R is also a stuttering bisimulation. By Lemma 4, so is R^+. Since $R \subseteq R^+$, $c R^+ d$ and $d R^+ e$. By the transitivity of R^+, we conclude $c R^+ e$, and hence $c \sim_{st} e$. □

Corollary 1. *$\sim_{st}$-correspondence is an equivalence relation.*

In the remainder of this section we establish that stuttering bisimilarity preserves deadlock and the validity of $\text{CTL}^*_{-\text{X}}$ formulas.

Theorem 2. *If $c \sim_{st} d$ then c has a deadlock iff d has a deadlock.*

Proof. Suppose c has a deadlock (when d has a deadlock the proof is similar). This means that there exist $c_0, \ldots, c_n \in C$ such that

$$c_0 = c,\ c_0 \to \cdots \to c_n,\ c_n \not\to \text{and } c_n \not\downarrow$$

By Lemma 2b, there exist $d_0, \ldots, d_m \in C$ such that

$$d_0 = d,\ d_0 \to \cdots \to d_m,\ d_m \not\to \text{and } c_n R d_m.$$

Suppose $d_m \downarrow$. Then, there exist $c_n^0, \ldots, c_n^k \in C$ such that

$$c_n^0 \to \cdots \to c_n^k, \text{ and } c_n^k \downarrow.$$

This is however not possible (even when $k = 0$) because $c_n \not\to \text{and } c_n \not\downarrow$ □

The following lemma plays a crucial role in the proof that stuttering bisimilarity preserves the validity of $\text{CTL}^*_{-\text{X}}$ formulas.

Lemma 5. *If $c \sim_{st} d$, then for every path from c there is a $\sim_{st}$-corresponding path from d.*

Proof. Let $c_0, c_1, c_2, \ldots$ be a path from c. There are two cases:

- if $c_0 \to \cdots \to c_n$, $c_n \not\to$ and $c_{i+1} = c_i$ for all $i \geq n$, then the statement follows directly from Lemma 2b;
- if $c_0 \to c_1 \to c_2 \to \cdots$, then the statement follows directly from Lemma 3.

□

Now we present the main theorem.

Theorem 3. *If $c \sim_{st} d$ then for all $\text{CTL}^*_{-\text{X}}$ formulas φ, $c \vDash \varphi$ iff $d \vDash \varphi$.*

Proof. The proof is a straightforward adaptation of the one of Theorem 3.2.3 in [18], replacing the calls to Lemma 3.2.2 in that proof by calls to Lemma 5. □

4 Stuttering Congruence

First we extend the definition of stuttering bisimilarity to the level of χ processes.

Definition 9. *Two processes p and q are stuttering bisimilar, denoted $p \sim_{st} q$, if for all $\sigma \in \Sigma$, $\langle p, \sigma \rangle \sim_{st} \langle q, \sigma \rangle$.*

To see that stuttering bisimilarity is not a congruence on χ processes, consider the following example.

Example 1. Note that the execution of a send action does not affect the state (see Rule 4 in Table 1), so $a!0 \sim_{st} b!0$. However, $a!0 \parallel a?x \not\sim_{st} b!0 \parallel a?x$, for the process on the left can do a communication action and change the value of x, whereas the process on the right cannot. It follows that $\sim_{st}$ is not a congruence for parallel composition. Also note that $a!0 \sim_{st} \mathit{skip}$, whereas $\partial(a!0) \not\sim_{st} \partial(\mathit{skip})$ ($\partial(a!0)$ is deadlocked; $\partial(\mathit{skip})$ does a τ-transition and terminates successfully). So $\sim_{st}$ is not a congruence for encapsulation either.

The example shows that for an equivalence on χ processes to be a congruence, it should not be completely action insensitive. We adapt the definition of stuttering bisimilarity in such a way that it distinguishes the send and receive actions from the other actions.

Let $\mathcal{A}^{com}$ be the set of all send and receive actions. Let $c \hookrightarrow c'$ mean that there is an $a \in \mathcal{A} \backslash \mathcal{A}^{com}$ such that $c \xrightarrow{a} c'$. Furthermore, let $c \xrightarrow{(a)} c'$ denote $c \xrightarrow{a} c'$ when $a \in \mathcal{A}^{com}$, and $c \hookrightarrow c'$ when $a \in \mathcal{A} \backslash \mathcal{A}^{com}$.

Definition 10. *A symmetric relation $R \subseteq C \times C$ is an* interaction sensitive stuttering bisimulation *iff, for all $(c, d) \in R$, c and d have the same state and:*

1. *if* $c\downarrow$, *then there exist* $d_0, \ldots, d_n \in C$ *such that*

$$d_0 = d,\ d_0 \hookrightarrow \cdots \hookrightarrow d_n,\ d_n\downarrow \text{ and } cRd_i \text{ for all } i \leq n,$$

2. *if* $c \xrightarrow{a} c'$, *then there exist* $d_0, \ldots, d_n \in C$ *such that*

$$d_0 = d,\ d_0 \hookrightarrow \cdots \hookrightarrow d_{n-1} \xrightarrow{(a)} d_n,\ cRd_i \text{ for all } i \leq n-1,\ \text{and } c'Rd_n$$

(we allow n *to be* 0 *only if* $a \in \mathcal{A}\backslash\mathcal{A}^{com}$ *and* c *and* c' *have the same state),*

3. *if there exists an infinite sequence* $c_0, c_1, c_2, \ldots \in C$ *such that*

$$c_0 = c,\ c_0 \hookrightarrow c_1 \hookrightarrow c_2 \hookrightarrow \cdots \text{ and } c_iRd \text{ for all } i \geq 0,$$

then there exist $d' \in C$ *and* $j > 0$ *such that* $d \hookrightarrow d'$ *and* c_jRd'.

Two configurations c *and* d *are* interaction sensitive stuttering bisimilar*, denoted* $c \sim_{isst} d$*, if there exists an interaction sensitive stuttering bisimulation* R *such that* cRd.

Theorem 4. *Interaction sensitive stuttering bisimilarity is an equivalence.*

Proof. Reflexivity and symmetry are proved as before. For the transitivity proof, it can be easily seen that Lemmas 1, 2a, and 3 hold when $\rightarrow$ is replaced by $\hookrightarrow$. Then, the proof goes similarly as for Theorem 1. □

To show that all the previous results hold, we prove the following theorem.

Theorem 5. *Interaction sensitive stuttering bisimilarity is a stuttering bisimulation.*

Proof. Suppose that $c \sim_{isst} d$. To show that the termination and transfer condition hold is trivial since $c \hookrightarrow c'$ implies $c \rightarrow c'$ for all $c, c' \in C$. We verify the divergence condition in detail.

Suppose $c_0 \rightarrow c_1 \rightarrow c_2 \rightarrow \cdots$ and $c_i \sim_{isst} d$ for all $i \geq 0$.

If every arrow in this sequence is a '$\hookrightarrow$' arrow, we have that there exist d' and $i > 0$ such that $d \hookrightarrow d'$ and $c_i \sim_{isst} d'$. In this case we are done because $d \hookrightarrow d'$ implies $d \rightarrow d'$.

Suppose now $c_0 \hookrightarrow \cdots \hookrightarrow c_n \xrightarrow{a} c_{n+1}$ and $a \in \mathcal{A}^{com}$.

Since $c_n \sim_{isst} d$, there exist $d_0, \ldots, d_m \in C$ such that $d_0 = d$,

$$d_0 \hookrightarrow \cdots \hookrightarrow d_{m-1} \xrightarrow{a} d_m, c_n \sim_{isst} d_i \text{ for all } i \leq m-1 \text{ and } c_{n+1} \sim_{isst} d_m.$$

Because $a \in \mathcal{A}^{com}$, $m > 0$. Then, $c_n \sim_{isst} d_1$ or $c_{n+1} \sim_{isst} d_1$. If $c_n \sim_{isst} d_1$ and $n = 0$ then, since $c_0 \sim_{isst} d_0$, $c_1 \sim_{isst} d_0$ and $\sim_{isst}$ is symmetric and transitive, we conclude that $c_1 \sim_{isst} d_1$. □

The following example shows that interaction sensitive stuttering bisimulation is still not a congruence.

Example 2. Note that $\mathit{skip} \sim_{\mathit{isst}} \mathit{skip}$ and $\delta \sim_{\mathit{isst}} \mathit{skip} \mathbin{;} \delta$. However, $\mathit{skip} \mathbin{[\!]} \delta \not\sim_{\mathit{isst}} \mathit{skip} \mathbin{[\!]} \mathit{skip} \mathbin{;} \delta$, for the right-hand side process can execute *skip* and then deadlock, while the left-hand side process never deadlocks. So, interaction sensitive stuttering bisimulation is not a congruence for alternative composition.

Both the problem illustrated in the above example, and its solution are well known; we need to add a root condition.

Definition 11. *Two configurations c and d are* rooted interaction sensitive stuttering bisimilar *(notation: $c \sim_{\mathit{riss}} d$) iff*

1. $c{\downarrow}$ *iff* $d{\downarrow}$
2. *if* $c \xrightarrow{a} c'$, *then there exists* $d' \in C$ *such that* $d \xrightarrow{(a)} d'$ *and* $c' \sim_{\mathit{isst}} d'$,
3. *if* $d \xrightarrow{a} d'$, *then there exists* $c' \in C$ *such that* $c \xrightarrow{(a)} c'$ *and* $c' \sim_{\mathit{isst}} d'$,

It is clear that $c \sim_{\mathit{riss}} d$ implies $c \sim_{\mathit{isst}} d$. Also, that rooted interaction sensitive stuttering bisimilarity is an equivalence relation is easily proved. The notion induces a congruence on χ processes.

Definition 12. *Two processes p and q are* stuttering congruent, *denoted $p \cong_{st} q$, if $\langle p, \sigma\rangle \sim_{\mathit{riss}} \langle q, \sigma\rangle$ for all $\sigma \in \Sigma$.*

Clearly, $\cong_{st}$ is an equivalence relation and $p \cong_{st} q$ implies $p \sim_{st} q$. We show that it is a congruence for the constructs of χ.

Theorem 6. *For all $p, q, \overline{p}, \overline{q} \in P$, if $p \cong_{st} q$ and $\overline{p} \cong_{st} \overline{q}$, then:*

1. $b :\!\rightarrow p \cong_{st} b :\!\rightarrow q$,
2. $p \mathbin{;} \overline{p} \cong_{st} q \mathbin{;} \overline{q}$,
3. $p \mathbin{[\!]} \overline{p} \cong_{st} q \mathbin{[\!]} \overline{q}$,
4. $p^* \cong_{st} q^*$,
5. $p \parallel \overline{p} \cong_{st} q \parallel \overline{q}$,
6. $[\![s \mid p]\!] \cong_{st} [\![s \mid q]\!]$ *for all states s,*
7. $\partial(p) \cong_{st} \partial(q)$.

5 Application

In [20], the translation from χ to PROMELA is discussed in detail. It is pointed out that the translation is straightforward for some constructs of χ (e.g., for assignments and alternative composition), since they also exist in PROMELA. However, the translation of guards, nested scopes and nested parallelism is less straightforward, since they have no direct equivalents in PROMELA. In the preprocessing phase of the proposed translation, nested scopes and certain occurrences of nested parallelism are eliminated, and guards are pushed down to the level of atomic processes. In this section we indicate how this preprocessing phase can be proved correct using the notion of stuttering congruence.

To be able to state some of the results pertaining to the scope operator, we first need to define the notion of *free* occurrence of a variable x in a process p. An occurrence of a variable x in a process p is called *free* if there is no subprocess of p of the form $[\![s \mid q]\!]$ with $x \in \mathrm{dom}(s)$ and q containing the occurrence of x. We denote by $\mathit{free}(p)$ the set of all variables with a free occurrence in p.

The following theorem explains how scope operators can be pulled out.

Theorem 7. *Let $p, q \in P$, let $s, s_1, s_2 \in \Sigma$ and let $b \in B$ such that $\mathrm{dom}(b) \cap \mathrm{dom}(s) = \mathit{free}(p) \cap \mathrm{dom}(s) = \mathit{free}(q) \cap \mathrm{dom}(s) = \emptyset$. Then:*

1. $b :\!\rightarrow [\![s \mid p]\!] \mathrel{\hat{=}}_{st} [\![s \mid b :\!\rightarrow p]\!]$,
2. $[\![s \mid p]\!] \,;\, q \mathrel{\hat{=}}_{st} [\![s \mid p \,;\, q]\!]$,
3. $p \,;\, [\![s \mid q]\!] \mathrel{\hat{=}}_{st} [\![s \mid p \,;\, q]\!]$,
4. $[\![s \mid p]\!] \mathbin{[\!]} q \mathrel{\hat{=}}_{st} [\![s \mid p \mathbin{[\!]} q]\!]$,
5. $[\![s \mid p]\!] \parallel q \mathrel{\hat{=}}_{st} [\![s \mid p \parallel q]\!]$,
6. $[\![s_1 \mid [\![s_2 \mid p]\!]]\!] \mathrel{\hat{=}}_{st} [\![\gamma(s_1, s_2) \mid p]\!]$,
7. $[\![s \mid \partial(p)]\!] \mathrel{\hat{=}}_{st} \partial([\![s \mid p]\!])$.

In case of a repetition, we need to be careful: $[\![s \mid p]\!]^*$ is not stuttering congruent with $[\![s \mid p^*]\!]$, for in $[\![s \mid p]\!]^*$ the state s is applied before each repetition of p, while in $[\![s \mid p^*]\!]$ it is only applied before the first repetition. The solution is to incorporate in $[\![s \mid p^*]\!]$ the effect of the state s after each repetition of p by a sequential composition of assignments of the form $x := s(x)$, one for every $x \in \mathrm{dom}(s)$. That is, if $\mathrm{dom}(s) = \{x_1, \ldots, x_n\}$, then we propose to replace $[\![s \mid p]\!]^*$ by $[\![s \mid (p \,;\, x_1 := s(x_1) \,; \ldots ; x_n := s(x_n))^*]\!]$. Note, however, that this only works if p does not have the option to terminate immediately (i.e., if $\langle p, \sigma\rangle \not\downarrow$ for all states σ). For, if $\langle p, \gamma(s, \sigma)\rangle\downarrow$, then $\langle [\![s \mid (p \,;\, x := c)^*]\!], \sigma\rangle$ can do the action $aa(x, c)$, whereas $\langle [\![s \mid p]\!]^*, \sigma\rangle$ cannot do the same action (unless p can do it), and thus the root condition is violated.

Let $\overline{P}$ be the set of processes that do not contain ε and in which every occurrence of the star operator is immediately followed by the sequential composition operator. Then $\langle p, \sigma\rangle \not\downarrow$ for all $\sigma \in \Sigma$ and all $p \in \overline{P}$, as is easily proved by structural induction on p.

Theorem 8. *Let $p \in \overline{P}$, let $x \in V$, and let $c \in C$. If s is a state such that $\mathrm{dom}(s) = \{x_1, \ldots, x_n\}$, then*

$$[\![s \mid p]\!]^* \mathrel{\hat{=}}_{st} [\![s \mid (p \,;\, x_1 := s(x_1) \,; \cdots ; x_n := s(x_n))^*]\!].$$

The next theorem explains how guards can be distributed over all operators except parallel composition.

Theorem 9. *Let $p, q \in P$, let $s \in \Sigma$ and let $b, b_1, b_2 \in B$, then:*

1. $\mathit{true} :\!\rightarrow p \mathrel{\hat{=}}_{st} p$
2. $b_1 :\!\rightarrow b_2 :\!\rightarrow p \mathrel{\hat{=}}_{st} (b_1 \wedge b_2) :\!\rightarrow p$
3. $b :\!\rightarrow (p \,;\, q) \mathrel{\hat{=}}_{st} (b :\!\rightarrow p) \,;\, q$
4. $b :\!\rightarrow (p \mathbin{[\!]} q) \mathrel{\hat{=}}_{st} (b :\!\rightarrow p) \mathbin{[\!]} (b :\!\rightarrow q)$

5. $b :\rightarrow p^* \cong_{st} (b :\rightarrow p)\,;p^* \,[\!]\, b :\rightarrow \varepsilon$
6. $b :\rightarrow \partial(p) \cong_{st} \partial(b :\rightarrow p)$

Note that the process $b :\rightarrow (p \parallel q)$ is not stuttering congruent with the process $(b :\rightarrow p) \parallel (b :\rightarrow q)$. For suppose the processes are executed in a state in which b is *true* and an action from p changes the state in such a way that the value of b becomes *false*. Then the process $b :\rightarrow (p \parallel q)$ proceeds as $p' \parallel q$ and the process $(b :\rightarrow p) \parallel (b :\rightarrow q)$ as $p' \parallel (b :\rightarrow q)$, and in the latter process the option q is blocked.

We finish this section with a theorem that allows us to simplify nested parallelism in the context of sequential composition. For similar reasons as before it is formulated in terms of the set $\overline{P}$.

Theorem 10. *Let $p, q, r \in \overline{P}$, and let $w \in V$ such that $w \notin free(p) \cup free(q) \cup free(r)$. Then,*

1. $(p \parallel q)\,;r \cong_{st} [\![w \shortmid\!\rightarrow 0 \mid p\,;w := w+1 \parallel q\,;w := w+1 \parallel (w=2) :\rightarrow r]\!]$,
2. $p\,;(q \parallel r) \cong_{st} [\![w \shortmid\!\rightarrow 0 \mid p\,;w := w+1 \parallel (w=1) :\rightarrow q \parallel (w=1) :\rightarrow r]\!]$.

6 Conclusion

In this paper we have proposed the notion of stuttering congruence for the modeling and simulation language χ. We have proved that if two χ processes are stuttering congruent, then, with respect to any state σ, they satisfy the same $\text{CTL}^*_{-\text{X}}$ formulas and either both have a deadlock or neither of them. Stuttering congruence is a behavioral congruence for the constructs of discrete-event, untimed part of χ, i.e., it is defined directly on the operational semantics of the language and it is a congruence for its constructions. Therefore, it is suitable for establishing the correctness of syntactic transformations on χ models.

We have illustrated the use of stuttering congruence in correctness proofs of syntactic transformations by indicating how (a part of) the preprocessing phase of the translation from χ to PROMELA can be proved correct. It is explained in detail in [20] that if a χ model satisfies a few general restrictions, then the preprocessing phase yields a χ model that can be straightforwardly translated into PROMELA. The resulting PROMELA specification can then be verified with SPIN. Incidentally, note that for Theorems 8 and 10 it is essential that stuttering congruence allows 'stuttering'; the transformations in these theorems do not preserve the validity of full CTL^*.

Currently, a translation from χ to UPPAAL is being developed, and it also involves a preprocessing phase. We think that stuttering congruence will be suitable for proving the correctness of that preprocessing phase too.

As future work, we mention the extension of the results obtained in this paper to full timed χ [19], proving the preservation of the validity of a timed variant of $\text{CTL}^*_{-\text{X}}$.

Acknowledgements. The authors would like to thank Jos Baeten for commenting on a draft of this paper, and the members of the TIPSy project for discussions.

References

1. S. Blom, W. Fokkink, J.F. Groote, I. van Langevelde, B. Lisser, and J.C. van de Pol. μCRL: A toolset for analysing algebraic specifications. In *Proceedings of CAV2001*, LNCS 2102, pages 250–254, 2001.
2. E. Bortnik, N. Trčka, A. J. Wijs, S. P. Luttik, J. M. van de Mortel-Fronczak, J. C. M. Baeten, W. J. Fokkink, and J. E. Rooda. Analyzing a χ model of a turntable system using SPIN, CADP and UPPAAL. *Journal of Logic and Algebraic Programming*, 2005. To appear.
3. V. Bos and J. J. T. Kleijn. Automatic verification of a manufacturing system. *Robotics and Computer Integrated Manufacturing*, 17:185–198, 2001.
4. V. Bos and J.J.T. Klein. *Formal specification and analysis of industrial systems.* PhD thesis, Eindhoven University of Technology, 2002.
5. M. C. Browne, E. M. Clarke, and O. Grumberg. Characterizing finite Kripke structures in propositional temporal logic. *Theor. Comput. Sci.*, 59:115–131, 1988.
6. E.J.J. van Campen. *Design of a multi-process multi-product wafer fab.* PhD thesis, Eindhoven University of Technology, 2000.
7. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model checking.* MIT Press, Cambridge, Massachusetts, 1999.
8. D. Dams. *Abstract Interpretation and Partition Refinement for Model Checking.* PhD thesis, Eindhoven University of Technology, 1996.
9. J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP - a protocol validation and verification toolbox. In *Proceedings 8th of CAV'96*, LNCS 1102, pages 437–440, 1996.
10. J.J.H. Fey. *Design of a fruit juice blending and packaging plant.* PhD thesis, Eindhoven University of Technology, 2000.
11. R. Gerth, R. Kuiper, D. Peled, and W. Penczek. A partial order approach to branching time logic model checking. *Information and Computation*, 150(2):132–152, 1999.
12. J. A. Govaarts. Efficiency in a lean assembly line: a case study at NedCar born. Master Thesis, 1997.
13. J. F. Groote and F. W. Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In M. S. Paterson, editor, *Proceedings of 17th ICALP*, LNCS 443, pages 626–638, 1990.
14. G.J. Holzmann. *The SPIN model checker.* Addison-Wesley, 2003.
15. K.G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.
16. B. Luttik and N. Trčka. Stuttering congruence for χ. Computer Science Report 05/13, Eindhoven University of Technology, 2005.
17. K. S. Namjoshi. A simple characterization of stuttering bisimulation. In *Proceedings of 17th FST & TCS*, LNCS 1346, pages 284–296, 1997.
18. R. De Nicola and F. W. Vaandrager. Three logics for branching bisimulation. *J. ACM*, 42(2):458–487, 1995.
19. R.R.H. Schiffelers, D.A. van Beek, K.L. Man, M.A. Reniers, and J.E. Rooda. Syntax and semantics of timed Chi. Computer Science Report 05/09, Eindhoven University of Technology, 2005.
20. N. Trčka. Verifying χ models of industrial systems with Spin. Computer Science Report 05/12, Eindhoven University of Technology, 2005.
21. D. A. van Beek, A. van der Ham, and J. E. Rooda. Modelling and control of process industry batch production systems. In *Proceedings of 15th Triennial World Congress of the International Federation of Automatic Control*, Barcelona, 2002.

Verifying Pattern-Generated LTL Formulas: A Case Study

Salamah Salamah[1], Ann Gates[1], Steve Roach[1], and Oscar Mondragon[2]

[1] Computer Science Department, University of Texas at El Paso,
El Paso TX 79902, USA
salamah, agates, sroach@cs.utep.edu,
[2] ESICenter, ITESM, Mexico
oscar.mondragon@itesm.mx,

Abstract. The Specification Pattern System (SPS) and the Property Specification (Prospec) tool assist a user in generating formal specifications in Linear Temporal Logic (LTL), as well as other languages, from property patterns and scopes. Patterns are high-level abstractions that provide descriptions of common properties, and scopes describe the extent of program execution over which the property holds. The purpose of the work presented in this paper is to verify that the generated LTL formulas match the natural language descriptions, timelines, and traces of computation that describe the pattern and scope. The LTL formulas were verified using the Spin model checker on test cases developed using boundary value analysis and equivalence class testing strategies. A test case is an LTL formula and a sequence of Boolean valuations. The LTL formulas were those generated from SPS and Prospec. The Boolean valuations of propositions in the LTL formula are generated by a deterministic, single-threaded Promela program that was run using the software model-checker Spin. For each pattern, a suite of test cases was. The experiments uncovered several errors in both the SPS-generated and the Prospec-generated formulas.

1 Introduction

Tools that simplify the creation of Linear Temporal Logic LTL formulas are of interest to the model checking community and others, but only if the resulting formulas match the intent of the specifier. This paper describes the verification of LTL formulas generated by two specification approaches: the Property Specification Patterns system (SPS)[2, 3] and Property Specification (Prospec)[10–12]. These two approaches use property pattern and scope to assist the specification of formal properties specified in LTL as well as other languages. Patterns are high-level abstractions that provide descriptions of common properties, and scopes describe the extent of program execution over which the property holds. Previous verification of the Prospec LTL formulas was obtained using the Spin model checker on variety of test cases, but without a disciplined strategy for test case selection. Previous verification of the SPS LTL formulas utilized peer reviewers and verification through Finite State Verification tools[14].

P. Godefroid (Ed.): SPIN 2005, LNCS 3639, pp. 200–220, 2005.

The verification described here used the software model checker Spin on test cases that were systematically developed based on equivalence class analysis and boundary value analysis. Equivalence class testing is a technique for minimizing the number of test cases by creating a single test case for each equivalence set of input values of a system and boundary value analysis is a method in which test cases are designed to include representatives of boundary values.

This use of Spin differs from traditional model checking. The model used here produces a simple finite state automaton with exactly one possible execution trace and only a few states. This model is used as a test bed for the analysis of LTL formulas. The simplicity of the model makes inspection feasible. The approach to validation of the generated LTL formulas is to create test cases for which the desired output is obvious, even if the meaning of the LTL formula is not immediately so. A test case is an LTL formula and a sequence of Boolean valuations. The LTL formulas were those generated from SPS and Prospec. The Boolean valuations of propositions in the LTL formula are generated by a deterministic, single-threaded Promela program that was run using Spin. For each pattern, a suite of test cases was developed using boundary value analysis and equivalence class testing strategies. The experiments uncovered several errors in both the SPS-generated and the Prospec-generated formulas.

This paper first presents a background on SPS and Prospec. These sections present the verbatim documentation that is made available to the user. The next section describes the testing process that was used to verify the formulas. Last, the results are described followed by a discussion. Appendix A gives the complete set of test cases for the *Absence-Before R* and *Precedence-Between L and R* patterns, respectively. Appendix B presents screen-shots of the Prospec tool, and Appendix C provides examples of failed test cases for both SPS and Prospec.

2 Background

2.1 Property Specification Patterns System

SPS defines patterns and scopes to assist the practitioner in formally specifying software properties. Patterns capture the expertise of developers by describing solutions to recurrent problems [4]. Each pattern describes the structure of specific behavior, defines the patterns relationship with other patterns, and defines the scope over which the property holds. The SPS website [14] provides descriptions of the patterns, including intent, relationships, and known uses. After the user selects a pattern and a specification language, the website displays a mapping for each scope in the chosen language. The target specification languages include: Linear Temporal Logic (LTL) [7], Computational Tree Logic (CTL) [6], and Graphical Interval Logic (GIL) [1, 5]. In the definitions given in this section, P, T, L, and R represent propositions.

The main patterns defined by SPS are: *universality*, *absence*, *existence*, *precedence*, and *response*. The descriptions given below are taken verbatim from the SPS website.

- *Absence*: To describe a portion of a system's execution that is free of certain events or states.
- *Universality*: To describe a portion of a system's execution which contains only states that have a desired property. Also known as Henceforth and Always.
- *Existence*: To describe a portion of a system's execution that contains an instance of certain events or states. Also known as Eventually.
- *Precedence*: To describe relationships between a pair of events/states where the occurrence of the first is a necessary pre-condition for an occurrence of the second. We say that an occurrence of the second is enabled by an occurrence of the first.
- *Response*: To describe cause-effect relationships between a pair of events/states. An occurrence of the first, the cause, must be followed by an occurrence of the second, the effect. Also known as Follows and Leads-to.

In SPS, each pattern is associated with a scope that defines the extent of program execution over which a property pattern is considered. There are five types of scopes defined in SPS, as shown in Figure 1

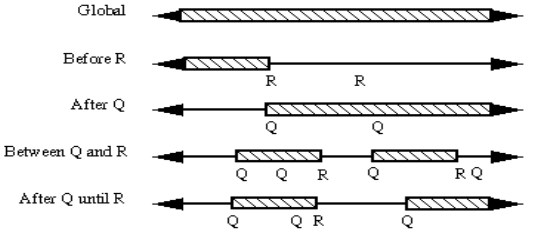


Fig. 1. Scopes in SPS [15]

The description of the scopes is described verbatim from the scopes section of the SPS website [15]:

> Each pattern has a scope, which is the extent of the program execution over which the pattern must hold. There are five basic kinds of scopes: global (the entire program execution), before (the execution up to a given state /event), after (the execution after a given state/event), between (any part of the execution from one given state/event to another given state/event) and after-until (like between but the designated part of the execution continues even if the second state/event does not occur). The scope is determined by specifying a starting and an ending state/event for the pattern: the scope consists of all states/events beginning with the starting state/event and up to but not including the ending state/event.

The figure above illustrates the portions of an execution that are designated by the different kinds of scopes. We note that a scope itself should be interpreted as optional; if the scope delimiters are not present in an execution then the specification will be true.

Before and after scopes for our patterns are interpreted relative to the first occurrence of the designated state/event. We have done this because it matches our experience with real specifications. Note, however, that we could just as easily interpret these scopes relative to the last occurrence of the designated state/event (the mappings given in the patterns are easily transformed to match this interpretation). At present we do not see the need for supporting both first and last occurrence scopes, but as we gain experience applying the patterns we may wish to extend scopes in this way.

Scope operators are not present in most specification formalisms (interval logics are an exception). Nevertheless, our experience strongly indicates that most informal requirements are specified as properties of program executions or segments of program executions. Thus a pattern system for properties should mirror this view to enhance usability.

To facilitate discussions about SPS and Prospec in this paper we use L to denote Q in the $After\ Q$, $Between\ Q\ and\ R$, and $After\ Q\ until\ R$ scopes.

2.2 Prospec

The Property Specification tool (Prospec)[10–12] builds on SPS by facilitating the identification of SPS patterns and scopes as well as validation of specifications. Prospec displays traces of computation to illustrate the subtle differences among different scopes and patterns, and it displays a decision tree to guide the user through a series of decisions to support the user in selecting an appropriate scope or pattern. Given a computation represented as a sequence of states, and a finite set of events E, a trace of computation is a list indicating, for each moment of time t, which events from the set E occur at t. Figure 4 in Appendix B shows the Prospec window for selecting a pattern and the traces of computation that are given to elucidate each pattern. Tables 1 and 2 give the characteristics for pattern and scope, respectively, that appear in the Prospec tool to assist the user in understanding the pattern and scope.

Prospec extends SPS by introducing a classification for defining sequential and concurrent behavior. This is accomplished by including composite propositions (CP) as shown in Figure 5 in Appendix B. The CP taxonomy [9, 12] categorizes and defines the structure of multiple propositions to capture sequential and concurrent behavior. The taxonomy can be used in the property elicitation and specification process to guide practitioners in formally specifying properties, eliminating the subtleties associated with multiple events and conditions. When relations have not been carefully analyzed, composite propositions can expose incompleteness or ambiguities.

CP defined as conditions are used to describe concurrency, and those defined as events are used to describe activation or synchronization of processes or actions. The formal syntax and semantics for CP classes is given in Mondragon and Gates [11]. CP can be used to define boundaries of scopes and patterns with multiple propositions. For instance, an ordered sequence can define the left boundary of an *after L* scope, and multiple events can define the cause part of a response pattern.

Table 1. Summary of characteristics for patterns in Prospec

PATTERN	CHARACTERISTICS
Absence of (P)	1) Event or condition P does not hold within the states defined by the scope of interest. 2) The *absence* property is also known as alarm.
Existence of (P)	1) Event or condition P holds at least once within the states defined by the scope of interest. 2) The *existence* property is also known as eventually.
Universality of (P)	1) Event or condition P holds in every state of the scope of interest. 2) The *universality* property is also known as safety or invariant.
(T) *Precedes* (P)	1) T holds before P holds, where T and P are events or conditions 2) T may hold several times before P holds 3) P does not hold before T holds 4) P does not hold at the same state at which T holds 5) If T holds, then P may or may not hold 6) If T holds, then T does not hold when P holds 7) The *precedence* property represents a cause-effect relation, where T denotes a cause and P denotes an effect 8) There is no effect P without a cause T 9) T precedes P is also known as T before P
(T) *Responds to* (P)	1) P must be followed by T, where P and T are events or conditions 2) Some T follows each time that P holds 3) The same state at which T holds may follow two or more states at which P holds 4) T may hold at the same state as P holds 5) If T holds, then P may or may not hold at a previous state 6) The *response* property represents a cause-effect relation, where P denotes a cause and T denotes an effect 7) If cause P holds, then at some future state effect T holds 8) T responds to P is also know as T follows P

2.3 LTL Formulas of SPS and Prospec

Table 3 presents the SPS LTL mappings for each pattern and scope combination and Table 4 presents those generated by Prospec. These are the formulas being

Table 2. Summary of characteristics for scopes in Prospec

SCOPE	CHARACTERISTICS
Global	1) The scope denotes the entire computation. 2) The scope includes all the states in the computation. 3) The interval defined by the scope occurs once in a computation
Before R	1) The scope denotes a subsequence of states or events (an interval) that begins with the start of computation and ends with the state or event immediately preceding the event or state at which R holds for first time in the computation. 2) The interval does not include the state or event associated with R. 3) The interval defined by the scope occurs once in a computation. 4) One or more events (conditions) may be associated with R; a condition is a proposition and an event is a change in value of the proposition from one state to the next.
After L	1) The scope denotes a subsequence of states or events (an interval) that begins with the first event or state at which L holds and ends with termination of computation. 2) The interval includes the state or event associated with L 3) The interval defined by the scope occurs once in a computation. 4) One or more events (conditions) may be associated with L; a condition is a proposition and an event is a change in value of the proposition from one state to the next.
Between L and R	1) The scope denotes a subsequence of states or events (an interval) that begins when L holds and ends with the state or event immediately preceding the event or state at which R holds. 2) Event or condition L must hold and, at a different event or state in the future, R must hold. 3) The interval includes the state or event associated with L 4) The interval does not include the state or event associated with R. 5) The interval defined by the scope may occur more than once in a computation. 6) Multiple intervals may be defined within an interval when L holds more than once before R holds 7) One or more events (conditions) may be associated with L and R
After L *Until* R	1) The scope denotes a subsequence of states or events (an interval) that begins when L holds and ends either with the state or event immediately preceding the event or state at which R holds, or begins when L holds and ends with the termination of computation. 2) The interval includes the state or event associated with L 3) The interval does not include the state or event associated with R 4) The interval may repeat during a computation. 5) If L holds and R does not hold, the interval ends with termination of a computation. 6) The interval defined by the scope may occur more than once in a computation. 7) Multiple intervals may be defined within an interval when L holds more than once before R holds 8) One or more events (conditions) may be associated with L and R

verified in this paper. It is important to note that the SPS formulas for the *absence* pattern were rewritten to facilitate the specification of sequences of events in Prospec. The SPS LTL formula for the *absence* pattern ($[](\neg P)$) holds if P does not hold at any state of the computation. If P is an event, then it is not possible to assert that P does not hold at any state of the computation. By using the LTL equivalence ($[]\ \mathrm{P} \equiv \neg \diamond (\neg P)$), the absence pattern is expressed as: $(\neg \diamond (P))$. The latter formula holds when it is not the case that P holds at some state of the computation. This formula provides a more intuitive specification of properties such as the absence of a sequence of events. In addition, the formulas for the *Precedence* pattern were rewritten to define a *strict-precedence* pattern, i.e., S *strictly precedes* P, where S and P are events or conditions. Unlike the SPS *precedence* pattern, S and P cannot hold at the same state in this pattern. While the *precedence* pattern ($\neg P\, W\, S$) holds even when S and P hold at the same state, the Prospec formula ($\neg P\, W\, (S \wedge \neg P)$) stays true to the SPS description that states "Precedence says that some cause precedes each effect." Use of *Strict Precedence* enforces that S and P cannot hold at the same state.

Prospec formulas for patterns with the $After$ scope differ from SPS formulas. While the SPS formula for the *universality* pattern within $After$ scope considers every state at which L holds, the Prospec formula only considers the first state at which L holds.

3 Verification Methodology

Tools that automatically generate specifications require assurance that the results are correct. According to the SPS website, the SPS specifications were verified by reviews conducted with experts in the language. The Prospec specifications were verified using the Spin model checker on a variety test cases, where a test case used the generated LTL formula and a sequence of Boolean valuations. The Boolean valuations of propositions in the LTL formula are generated by a deterministic, single-threaded Promela program that was run using Spin. For each pattern, a suite of test cases was developed. The work presented in this paper followed the Prospec approach to verifying the formulas, but it used a more systematic approach to defining the test cases by incorporating boundary value analysis and equivalence class testing strategies.

3.1 Test Process

Each pattern is associated with a test suite. The test suite consists of a set of test cases for each scope, where a test case is a set of assignments for the propositions in the LTL formula under test such that a particular trace of computation is created. In other words, each test case will result in a particular trace of computation for an LTL formula associated with a pattern and scope combination. The test suites for both SPS and Prospec are identical, although for some patterns and scopes the expected results may differ.

Table 3. SPS' LTL formulas for pattern and scope

PATTERN	SCOPE	LTL Formula
Absence	*Global*	$\Box(\neg P)$
	Before R	$\Diamond R \rightarrow (\neg P U R)$
	After L	$\Box(L \rightarrow \Box(\neg P))$
	Between L and R	$\Box((L \wedge \neg(R) \wedge \Diamond R) \rightarrow (\neg(P) U R))$
	After L Until R	$\Box((L \wedge \neg(R)) \rightarrow (\neg(P) W R))$
Existence	*Global*	$(\Diamond P)$
	Before R	$\neg R W (P \wedge \neg R)$
	After L	$\Box(\neg L) \vee \Diamond(L \wedge \Diamond P))$
	Between L and R	$\Box((L \wedge \neg(R)) \rightarrow (\neg(R) W (P \wedge \neg R)))$
	After L Until R	$\Box((L \wedge \neg(R)) \rightarrow (\neg(R) U (P \wedge \neg R)))$
Universality	*Global*	$\Box P$
	Before R	$\Diamond R \rightarrow (P U R)$
	After L	$\Box(L \rightarrow \Box(P))$
	Between L and R	$\Box((L \wedge \neg(R) \wedge \Diamond R) \rightarrow (P U R))$
	After L Until R	$\Box((L \wedge \neg(R)) \rightarrow (P W R))$
Presedence	*Global*	$\neg(P) W T$
	Before R	$\Diamond R \rightarrow (\neg(P) U (T \vee R))$
	After L	$\Box \neg L \vee \Diamond(L \wedge (\neg P W T))$
	Between L and R	$\Box((L \wedge \neg(R) \wedge \Diamond R) \rightarrow (\neg(P) U ((T \vee R)))$
	After L Until R	$\Box(L \wedge \neg(R) \rightarrow (\neg(P) W (T \vee R)))$
Response	*Global*	$\Box(P \rightarrow \Diamond T)$
	Before R	$\Diamond R \rightarrow ((P \rightarrow (\neg(R) U (T \wedge \neg(R)))) U R)$
	After L	$\Box(L \rightarrow \Box(P \rightarrow \Diamond T))$
	Between L and R	$\Box((L \wedge \neg(R) \wedge \Diamond R) \rightarrow (P \rightarrow (\neg(R) U (T \wedge \neg(R)))) U R$
	After L Until R	$\Box((L \wedge \neg(R) \rightarrow (P \rightarrow (\neg(R) U (T \wedge \neg(R)))) W R)$

The Promela code consists of a do-loop that begins with the initial value of i set to zero and terminates when i reaches a predefined value called *limit*. Setting the value of *limit* to 20 supports construction of a variety of test cases that are dependant on the value of i to create a desired trace of computation. The Promela Code follows:

```
#define limit 20
byte i = 0;
active proctype seq( ){
  do
  ::  (i < limit) -> i = i+1;
  ::  (i == limit) -> break;
  od; }
```

Execution of a test case consists of assigning conditions to propositions, defining an LTL formula, and model checking the Promela code. The Promela code remains the same in all test cases. Assignments of conditions to propositions are done in the Xspin 3.4.2's LTL Property Manager. For example, in the test case where L is true in the second state, P is true in the seventh state, and R is true

Table 4. Prospec's LTL formulas for pattern and scope

PATTERN	SCOPE	LTL Formula
Absence	*Global*	$\neg(\diamond P)$
	Before R	$\diamond R \rightarrow \neg(\neg(R)UP)$
	After L	$\neg(L)W(L \wedge \neg(\diamond P))$
	Between L and R	$[](L \wedge \neg(R) \wedge \diamond R) \rightarrow \neg(\neg(R)UP))$
	After L Until R	$[](L \wedge \neg(R) \rightarrow \neg(\neg(R)UP))$
Existence	*Global*	$(\diamond P)$
	Before R	$\diamond R \rightarrow (\neg(R)U(P \wedge \neg(R)))$
	After L	$\neg(L)W(L \wedge (\diamond P))$
	Between L and R	$[](L \wedge \neg(R) \wedge \diamond R) \rightarrow (\neg(R)U(P \wedge \neg(R))))$
	After L Until R	$[](L \wedge \neg(R)) \rightarrow (\neg(R)U(P \wedge \neg(R))))$
Universality	*Global*	$[]P$
	Before R	$\diamond R \rightarrow (PUR)$
	After L	$\neg(L)W(L \wedge []P)$
	Between L and R	$[](L \wedge \neg(R) \wedge \diamond R) \rightarrow (PUR))$
	After L Until R	$[](L \wedge \neg(R)) \rightarrow (PWR))$
Presedence	*Global*	$\neg(P)WT$
	Before R	$\diamond R \rightarrow (\neg(P)U(T \vee R))$
	After L	$\neg(L)W(L \wedge (\neg(P)W(T)))$
	Between L and R	$[](L \wedge \neg(R) \wedge \diamond R) \rightarrow (\neg(P)U((T \vee R)))$
	After L Until R	$[](L \wedge \neg(R) \rightarrow (\neg(P)W(T \vee R)))$
Response	*Global*	$[](P \rightarrow \diamond T)$
	Before R	$\diamond R \rightarrow ((P \rightarrow (\neg(R)U(T \wedge \neg(R))))UR)$
	After L	$(\neg L)W(L \wedge [](P \rightarrow \diamond T))$
	*Between L and*R	$[]((L \wedge \neg(R) \wedge \diamond R) \rightarrow (P \rightarrow (\neg(R)U(T \wedge \neg(R))))UR$
	After L Until R	$[]((L \wedge \neg(R) \rightarrow (P \rightarrow (\neg(R)U(T \wedge \neg(R))))WR)$
Strict Presedence	*Global*	$\neg(P)W(T \wedge \neg(P))$
	Before R	$\diamond R \rightarrow (\neg(P)U((T \wedge \neg(P)) \vee R))$
	After L	$\neg(L)W(L \wedge (\neg(P)W(T \wedge \neg(P))))$
	Between L and R	$[](L \wedge \neg(R) \wedge \diamond R) \rightarrow (\neg(P)U((T \wedge \neg(P)) \vee R)))$
	After L Until R	$[](L \wedge \neg(R) \rightarrow (\neg(P)W((T \wedge \neg(P)) \vee R)))$

in the twelfth state, the following definitions are made: # define L (i == 1), # define P (i == 6), and # define R (i == 11). Figure 2 presents an example test case for the pattern *Existence*(P) with scope *Between* L and R. In this example, the test number is the number of the test in the test suite. The documentation includes a representation of the trace of computation to be tested. For example, in the trace "- - - - - L - - - - - P - - - R - - - - -" each character in the string represents a state. A dash (-) implies that none of the propositions is *true* at that state, a letter symbol, e.g., L, P, and R in this example, denotes that the proposition is *true* in the designated state. For example, in the trace above, L holds in state 5, and P holds in state 11. Displaying two letters between parenthesis implies that both propositions are valid at that state. For instance, in trace "- - (pq) - - r ", p and q both hold in state 2. The test case

documentation also includes the name of the pattern and scope being tested as well as the LTL formula that was given by SPS or generated from Prospec.

The LTL formula is tested by the Spin model checker. Because Spin does not support the weak until (W) operator, the LTL equivalence was used instead. For example, $aWb \equiv []a \vee (aUb)$. Next, the documentation gives the assignment for the propositions in the formula. These assignments are based on variable i in the Promela code and are selected so that the trace of computation is generated. In the example in Figure 2, L is defined as (i == 3), i.e., proposition L holds when variable i becomes 3 within the iteration. The Interval designation is used to document whether an interval(s) can be built in the trace of computation given by the test cases. This is relevant for all scopes except *Global*. Last, the test case documentation shows the expected results based on the natural language description and timelines provided by the respective system or tool and the actual results that were returned by the Spin model checker.

Test 2 : - - -L - - P - - R- - - - -
Pattern : Existence (P)
Scope : Between L and R
Formula: [] ((L && $(\neg R)\&\&(\diamond R)) \rightarrow (\neg((\neg R)U(P\&\&(\neg R)))))$
L: (i == 3)
P: (i == 6)
R: (i == 9)
Interval: Yes
Expected Result: No violation
Actual Result: No violation

Fig. 2. Sample Test Case

3.2 Equivalence Class Analysis

Equivalence class analysis partitions test cases into sets from which exemplars are selected. It assumes that test cases from a partition are equally likely to expose an error and, hence, only a representative test case is needed. This enables developers to significantly reduce the number of tests to run. In other words, the goal of equivalence class testing to minimize the number of possible tests while at the same time have enough coverage to provide confidence in the correctness of the code or system being tested. For example, the two traces of computation "L - P - - - - - -" and "L - - - - - - P - -" belong to the same equivalence class in the suite $existence(P)$ - $After\ L$ because P occurs after L. The test sets for the $Absence\ (P)$ and $Existence\ (P)$ patterns were partitioned at the upper level into two sets: 1) P does not hold in any state of the interval(s) and 2) P holds in some state of the interval(s). The test sets for the T *precedes* P (T *responds* to P) scopes were partitioned at the upper level into two sets: 1) T does not precede P (T does not respond to P) in any state of the interval(s) and 2) T

precedes P (T responds to P) in the interval (s). Note that the partitions for the *Universality* pattern and for *Global* scope are slight variations of the above partitions. The next level of partitions is based on the intervals defined by the scope as follows:

- *Before* R (*After* L)-Partition 1:
  - R (L) holds in the first state
  - R (L)holds in the last state
  - R (L)holds in other states
  - the interval is not built
- *Before* R (*After* L)-Partition 2:
  - L holds in the first state (not applicable for R)
  - R (L) holds in the last state
  - R (L) holds in other states
- *Between* L *and* R (*After* L *Until* R)-Partition 1:
  - the interval is not made
  - a single interval is made and
    - L holds in the first state
    - R holds in the last state
  - multiple intervals are made
  - nested intervals are made
- *Between* L *and* R (*After* L *Until* R)-Partition 2:
  - a single interval is made and
    - L holds in the first state
    - R holds in the last state
    - L and R hold in other states
  - multiple intervals are made
  - nested intervals are made

3.3 Boundary Value Analysis

The test cases defined in each of the sub-partitions above are based on boundary value analysis strategies. The boundary value analysis strategy is a method by which values are chosen to lie on data extremes. Example values are those representing the maximum value, minimum value, and the values just before and after boundaries. The idea is that, if a the system works correctly for these special values then it will work correctly for all values in between. For example, consider the following three test cases that are associated with *Absence* - *After* L and the sub-partition "L holds in other states" under Partition 2.

1. - - - - - - - - - P L - - - - - - - - - -
2. - - - - - - - - - - (LP) - - - - - - - - - -
3. - - - - - - - - - - L P - - - - - - - - -

Test case 1 is valid and the last two test cases are not valid. Test case 2 checks that P holds in the first state in which L becomes true. Test case 2 is also used for a $Before\ R$ test case, where L is replaced by R. In this case, the test would be valid. Tables 5 and 6 in Appendix A give the complete set of test cases for the *Absence-Before R* and *Precedence-Between L and R* patterns respectively. The complete set of test cases for all patterns and scopes combinations can be found in Salamah [13].

Test cases that check conditions that can be verified through other test cases were eliminated. For example, in the *T precedes P* pattern for the *Between L and R* scope, the following test case was eliminated: $---L(PT)---R$. There are two things that are being checked by this test case: it checks that strict precedence is upheld, and it checks that the interval is still built when R occurs in the last state of the computation. These two conditions are covered by test cases 2(a)(vi) and 2(b)(iii) as given in Table 6 of Appendix A.

4 Results and Discussion

Of the 25 formulas created using SPS, 16 were verified. Of the 30 formulas created using Prospec, 27 were verified based on interpretation of the documentation provided with each of these tools. Since the documentation is written in English, the potential for ambiguity is always present. Prospec has now been modified to generate formulas that align with the documentation. The following subsections discuss the errors detected by the verification effort and suggests ways to align SPS formulas to the documentation.

4.1 Prospec Errors

The testing of the Prospec generated formulas revealed errors in three formulas, all of which are in the *Absence* (p) pattern. The affected scopes are: $Before\ R$, *Between L and R*, and $After\ L\ until\ R$. The test cases, which identified the errors, created an execution trace in which P holds at the same state as the right boundary R. In each case, Spin reported a violation. According to Prospec's documentation, scopes consist of all the states beginning with, and including, the left boundary and ending with, but *not* including, the right boundary. Figure 3 shows an example test case where a Prospec's generated formula failed. The complete list is included in Appendix C.

To address the Prospec's errors, the affected formulas were changed as follows:

- *Absence Before R*: $\diamond R \rightarrow (\neg(\neg(R)U(P \wedge \neg R)))$
- *Absence Between L and R*: $[]((L \wedge (\neg R) \wedge (\diamond R)) \rightarrow (\neg(\neg(R)U(P \wedge \neg R))))$
- *Absence After L until R*: $[]((L \wedge (\neg R)) \rightarrow (\neg(\neg(R)U(P \wedge \neg R))))$

The formulas were tested with the test cases from the *Absence* suite. No violation were reported.

4.2 SPS Errors

The testing of the SPS generated formulas revealed errors in 9 formulas due to ambiguity of the documentation. The formulas that had questionable behavior were those generated from the scopes *Between Q and R* and *After Q until R* for the *Existence* and *Precedence* patterns. In addition, the formulas for the *Response* pattern in all the scopes did not behave as expected.

In the *Between Q and R* and *After Q until R*, the documentation implies that the proposition must hold only in the interval built by the first occurrence of Q and R, and not in the subintervals within the outer interval. Examination of Figure 1 for the *After Q* timeline, given in Section 2.1, shows two Qs in the interval. More importantly, the SPS documentation states: "Before and after scopes for our patterns are interpreted relative to the first occurrence of the designated state/event." This implies that the second Q is not being considered. Because the timelines for the *Between Q and R* and the *After Q until R* scopes are similar to the diagram for the *After Q* scope, one would expect the same interpretation of these scopes to hold. There is not a detailed description of these scopes to indicate otherwise. Inspection of the formulas for the *Absence*, *Universality*, and *Response* patterns in the *After Q* scope given in Table 3, however, shows that the pattern is checked after each occurrence of Q and not just the first occurrence of Q. A list of violated test cases in Appendix C show that the *existence* and *precedence* patterns in the *Between L and R* and *After L until R* are also being checked in all subintervals that are formed. For example; in the *existence* of *P Between L and R* the test "- - - L - - - P - - - L - - - R - -" returns a violation, even though the documentation of SPS indicates that the occurrence of one P within the outer interval should be enough; however, a violation is returned since P does not hold in the second interval. The only way for a user to know that the pattern is checked within subintervals is to examine the complex formulas that are generated. We argue that the intent of SPS and Prospec is to assist professionals, who may not be experts in a particular language, in creating formal specifications. Two approaches can be used to correct this: clarify the documentation to reflect the actual behavior of the formula, or revise the formula.

The ambiguity in the documentation concerning the behavior of the Response pattern can also lead to errors. The "Intent" section for the Response pattern as provided in Section 2.1 states that the intent of the Response pattern is "To describe cause-effect relationships between a pair of events/states. An occurrence of the first, the cause, must be followed by an occurrence of the second, the effect. Also known as Follows and Leads-to." In addition, the "Relationship" section of the Response pattern in the SPS website[14] states: "...Response says that some effect follows each cause." The English interpretation of the word "follows" according to Merriam Webster [8], is "to come or take place after in time, sequence, or order." The generated formula allows for S and P to occur in

the same state in the case of S responds to P, as validated by test cases shown in Appendix C. In LTL this is a valid interpretation of "follows" since the current state is considered part of the future. This subtlety will be missed by someone who is not an expert in LTL and, as a result, should be clearly stated.

Prospec Failed Test	**SPS Failed Test**
Test 6:	Test 26:
- - - - - (RP) - - - - - - - - - - - - - - -	- - - - - L - - - P - - L - - - R - - - -
Pattern : Absence (p)	Pattern : Existence (p)
Scope : Before R	Scope : Between L and R
Formula: $\diamond R \rightarrow \neg(\neg(R)UP)$	Formula: $[](\mathrm{L} \wedge(\neg R) \rightarrow (\neg(R)W(P \wedge \neg R)))$
P : (i == 5)	P : (i == 9)
R : (i == 5)	R : (i == 16)
	L : (i == 5) $\vee(i == 12)$
Interval: Yes	Interval: Yes
Expected Result: No violation	Expected Result: No violation
Actual Result: Violation	Actual Result: Violation

Fig. 3. Examples of Prospec and SPS Failed Tests

4.3 Discussion

It is critical to provide assurance concerning the correctness of formulas that are generated automatically from tools. Verifying complex formulas is difficult even when experts are involved, and Spin is effective at solving this problem. By defining a systematic approach for creating test cases and using a software model checker, it is possible to provide the user with an added layer of confidence that the formulas generated from SPS and Prospec are correct.

While the problems found in Prospec were related to an error in the formula, those in SPS are related to the ambiguity of the language in the documentation. An unexpected outcome from this experiment came from analysis of the difference in formulas generated by SPS and Prospec for the $After\ Q$ scope. As mentioned before, the $After\ Q$ scope considers the execution after a given state or event. Although the tests are able to show that the formulas generate expected answers, the behavior of the SPS and Prospec formulas is different. Prospec formulas consider only the first occurrence of the condition or event assigned to the left boundary (L) while SPS formulas take into consideration every occurrence of the condition or event assigned to the left boundary (Q). The use of SPS formulas for this scope may cause performance issues as unnecessary checks for additional Qs would result in delays in the model checkers executions. The situation described above provides an area of opportunity for the Spin community by considering the possibility to run a Promela code with two LTL formulas and letting the user know the formula with a better performance.

Acknowledgments

This research was partially supported by Consejo Nacional de Ciencia y Tecnologia under Contract 68761, NASA grant NCC5-205, and the NSF EAR-0225670 ITR GEON grant.

The authors would like to thank the authors of SPS and Prospec for their efforts in promoting the use of formal methods with practitioners. This paper was inspired mainly by these efforts.

References

1. Dillon, L., G. Kutty, L. E. Moser, P. M. Melliar – Smith, and Y.S. Ramakrishna, *A Graphical Interval Logic for Specifying Concurrent Systems*, ACM Transactions on Software Eng. and Methodology, **3** (1994), 131–165.
2. Dwyer, M. B., G. S. Avrunin, and J. C. Corbett, "Property Specification Patterns for Finite–State Verification," *Proceedings of the 2nd Workshop on Formal Methods in Software Practice*. Clearwater Beach, Florida, 1998, 7–15.
3. Dwyer, M. B., G. S. Avrunin, and J. C. Corbett, "Patterns in Property Specification for Finite–State Verification," *Proceedings of the 21st Intl. Conference on Software Engineering*, Los Angeles, CA, USA, 1999, 411–420.
4. Gamma, E. and R. Helm, Design Patterns, Elements of Reusable Object– Oriented Software, Addison–Wesley, 1995, 416.
5. Kutty, G., "A Graphical Environment for Temporal Reasoning," Dissertation, Electrical and Computer Engineering Department, University of California at Santa Barbara, 1994.
6. Laroussinie, F. and P. Schnoebelen, "Specification in CTL+Past for Verification in CTL," *Information and Computation,* 2000, 236–263.
7. Manna, Z. and A. Pnueli, "Completing the Temporal Picture," *Theoretical Computer Science,* 83(1), 1991, 97-130.
8. Merriam Webster Online , http://www.m-w.com/cgi-bin/dictionary?book=Dictionaryva=follow, April 2005.
9. Mondragon, O., A. Gates, and S. Roach, "Composite Propositions: Toward Support for Formal Specification of System Properties," *Proceedings of the 27th Annual IEEE/NASA Goddard Software Engineering Workshop,* Greenbelt, MD, USA, December 2002.
10. Mondragon, O., A. Q. Gates, and S. Roach, "Prospec: Support for Elicitation and Formal Specification of Software Properties," in O. Sokolsky and M. Viswanathan (Eds.): *Proceedings of Runtime Verification Workshop, ENTCS*, 89(2), 2004.
11. Mondragon, O. and A. Q. Gates, "Supporting Elicitation and Specification of Software Properties through Patterns and Composite Propositions," *Intl. Journal Software Engineering and Knowledge Engineering,*XS 14(1), Feb. 2004.
12. Mondragon, O., "Elucidation and Specification of Software Properties through Patterns and Composite Propositions to Support Formal Verification Techniques," Dissertation, The University of Texas at El Paso, May 2004.
13. Salamah, S., "Supporting Documentation for the SPS-Prospec Case Study," UTEP-CS-05-14, the University of Texas at El Paso, April 2005.
14. Spec Patterns, http://patterns.projects.cis.ksu.edu/, April 2005.
15. Spec Patterns, http://patterns.projects.cis.ksu.edu/documentation/patterns/scopes.shtml, April 2005.

Appendix

Appendix A. EQUIVALANCE CLASSES

A complete list of all the equivalence classes for all patterns and scopes used in the case study can be found in Salamah [13].

Table 5. Equivalance Classes for Absence-Before R

EQUIVALENCE CLASSES ON P	TEST CASES WITH BOUNDARY ANALYSIS AND EQUIVALENCE CLASSES ON R	EXPECTED RESULTS
P Does not hold in any state in the interval	1. R holds in first state:	
	a) R -	1a. Valid
	b) (RP) -	1b. Valid
	c) R P - - - - - - - - - - - - - - - - - - -	1c. Valid
	2. R holds in last state:	
	a) - R	2a. Valid
	b) - (RP)	2b. Valid
	3. R holds in other states:	
	a) - - - - - - - - R - - - - - - - - - - - -	3a. Valid
	b) - - - - - - - - - (RP) - - - - - - - - - - -	3b. Valid
	c) - - - R - - - - - - - P R - - - - - - - -	3c. Valid
	4. Interval is not built	
	a) -	4a. Valid
	b) - - - - - - - P - - - - - - P - - - - - -	4b. Valid
P holds in some state of the interval	5. R holds in last state:	
	- P R	5. Not valid
	6. R holds in other states:	
	a) - - - - - - - - - P R - - - - - - - - - -	6a. Not valid
	b) - - - - - - - - - - R P - - - - - - - - -	6b. Valid

Table 6. Equivalance Classes for Precedence Between L and R

EQUIVALENCE CLASSES ON P	TEST CASES WITH BOUNDARY ANALYSIS AND EQUIVALENCE CLASSES ON R	EXPECTED RESULTS
T Does not precede P in any state in the interval	1. The interval is not made:	
	a) - - - - - - - - - P - - - - - - - - - R	1a. Valid
	b) - - - - (LRP) - - - - - - - - - - - - - - - -	1b. Valid
	c) - - - - L - - - - - - - P - - - - - - - -	1c. Valid
	d) - - - - - - - - - - - - P - - - - - - - -	1d. Valid
	e) R - - - - - - - P - - - - L - - - - - - -	1e. Valid
	2. A single interval is made:	
	a) L holds in first state:	
	i. L - - - P - - - - R - - - - - - - - - - -	2ai. Not valid
	ii. L - - - T - - - - R - - - - - - - - - - -	2aii. Valid
	iii. L - - P - - T - - R - - - - - - - - - - -	2aiii. Not valid
	iv. L - - - - (RP) - - - - - - - - - - - - - - -	2aiv. Valid
	v. L - - - - - R P - - - - - - - - - - - - -	2av. Valid
	vi. L - - - (PT) - - - - R - - - - - - - - - - -	2avi. Not valid
	b) R holds in last state:	
	i. - - - - - - - - - - - P L - - - - - - - R	2bi. Valid
	ii. - - - - - - - - - - - - - - - - - L - - - (RP)	2bii. Valid
	iii. - - - - - - - - - - - L - - P - - - - - R	2biii. Not valid
	iv. - - - - - - - - - - - L - - P - - T - - R	2biv. Not valid
	v. - - - - - - - - - - - L - - T - - - - - R	2bv. Valid
	3. Multiple intervals are made:	
	a) L - - - - R - - - - - L - - - P - - R - -	3a. Not valid
	b) L - - - - R - - P - - - L - - - - R - - -	3b. Valid
	4. Nested intervals are made:	
	a) - - - - L - - - - - - - L - - P - - R - - -	4a. Not valid
	b) - - - - L - - P - - - L - - - - - R - - -	4b. Not valid
	c) - - - L - - - L - - - R - - P - - R - - -	4c. Valid
	d) - - L - - T - - L - - - P - - - - R - - -	4d. Valid in SPS Not valid in Prospec
T precedes P in the interval	5. A single interval is made:	
	a) L holds in first state:	
	i. L T - P - - - - - R - - - - - - - - - - -	5ai. Valid
	ii. (LT) P - - - - R - - - - - - - - - - - - -	5aii. Valid
	iii. L - - - T - P R - - - - - - - - - - - -	5aiii. Valid
	b) R holds in last state:	
	i. - - - - - - - - - - - - - - - L T P - - - R	5bi. Valid
	ii. - - - - - - - - - - - - - - - (TL) P - - - - R	5bii. Valid
	iii. - - - - - - - - - - - - - - - L - - T - P R	5biii.Valid
	c) L-R hold in other states:	
	i. - - - - - - - - L - - T - - P - - R - - -	5ci. Valid
	ii. - - - - - - - - - L - - T - - P - R - - -	5cii. Valid
	6. Multiple intervals are made:	
	a. L - T - P - - R - - - - L - - R - - L P R	6a. Not valid
	b. L - T P - - R - L - - T - P - R - L - P -	6b. Valid
	7. Nested intervals are made:	
	- - - L - - T - - L - - - P - R - - - - -	7. Valid in SPS Not valid in Prospec

Appendix B. Prospec's Screen-Shots

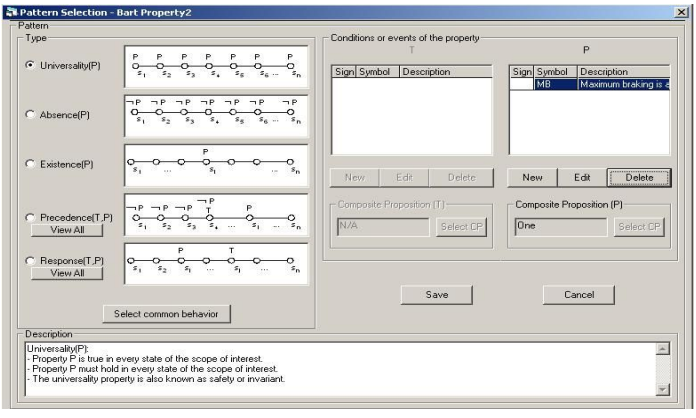


Fig. 4. Prospec's Pattern Screen.

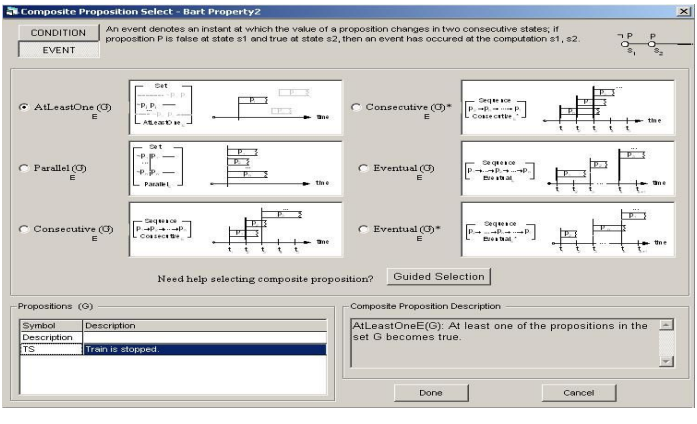


Fig. 5. Prospec's Composite Proposition Screen

Appendix C. SPS and Prospec's Sample Failed Tests

A complete list of all the test cases used in the case study can be found in Salamah [13].

Prospec Failed Test
Test 6 : - - - - - (RP) - - - - - - - - - - - - - - -
Pattern : Absence (p)
Scope : Before R
Formula: $\diamond R \rightarrow (\neg(\neg(R)UP))$
P : (i == 5)
R : (i == 5)
Interval: Yes
Expected Result: No violation
Actual Result: Violation
Test 26 : - - - - - L - - - - - - - - - - (RP) - - - -
Pattern : Existence (p)
Scope : Between L and R
Formula: $[](($L $\wedge(\neg R) \wedge (\diamond R)) \rightarrow (\neg(\neg(R)UP))$
P : (i == 16)
R : (i == 16)
L : (i == 5)
Interval: Yes
Expected Result: No violation
Actual Result: Violation
Test 46 : - - - - - L - - - - - - - - - - (RP) - - - -
Pattern : Existence (p)
Scope : After L Until R
Formula: $[](\text{L} \wedge(\neg R) \rightarrow (\neg(\neg(R)UP))$
P : (i == 16)
R : (i == 16)
L : (i == 5) $\vee(i == 12)$
Interval: Yes
Expected Result: No violation
Actual Result: Violation

Fig. 6. Examples of Prospec Failed Tests

Test 30 - - L - - - P - - - L - - - - - - R - - - Pattern : Existence (p) Scope : Between L and R Formula: $[](($L $\wedge(\neg R)) \rightarrow (\neg RW(P \wedge \neg R)))$ P : (i == 6) L : (i == 2) $\vee(i == 10)$ R : (i == 17) Interval: Yes Expected Result: No violation Actual Result: Violation	Test 42 - - L - - - P - - - L - - - - - - R - - - Pattern : Existence (p) Scope : After L Until R Formula: $[](($L $\wedge(\neg R)) \rightarrow (\neg RU(P \wedge \neg R)))$ P : (i == 6) L : (i == 2) $\vee(i == 10)$ R : (i == 17) Interval: Yes Expected Result: No violation Actual Result: Violation
Test 38 - - L - - - T - - - L - - P - - - R - - - Pattern : (T) Precedes (p) Scope : Between L and R Formula: $[](($L $\wedge(\neg R) \wedge \diamond R) \rightarrow (\neg PU(T \vee R)))$ P : (i == 13) L : (i == 2) $\vee(i == 10)$ R : (i == 17) T : (i == 6) Interval: Yes Expected Result: No violation Actual Result: Violation	Test 41 - - L - - - T - - - L - - P - - - R - - - Pattern : (T) Precedes (p) Scope : After L until R Formula: $[](($L $\wedge(\neg R)) \rightarrow (\neg PW(T \vee R)))$ P : (i == 13) L : (i == 2) $\vee(i == 10)$ R : (i == 5) T : (i == 6) Interval: Yes Expected Result: No violation Actual Result: Violation
Test 3 - - - - - (TP) - - - - - - - - - - - - - - - Pattern : T Responds to P Scope : Global Formula: $[](P \rightarrow \diamond T)$ P : (i == 5) T : (i == 5) Interval: Yes Expected Result: Violation Actual Result: No violation	Test 9 - - - - - (TP) - - R - - - - - - - - - - - - Pattern : T Responds to P Scope : Before R Formula: $\diamond R \rightarrow (P \rightarrow (\neg RU(T \wedge \neg R)))UR$ P : (i == 5) T : (i == 5) R : (i == 8) Interval: Yes Expected Result: Violation Actual Result: No violation
Test 20 - - L - - (RP) - - - - - - - - - - - - - - - Pattern : T Responds to P Scope : After L Formula: $[](L \rightarrow [](P \rightarrow \diamond T))$ P : (i == 5) T : (i == 5) L : (i == 2) Interval: Yes Expected Result: Violation Actual Result: No violation	Test 36 - - L - - (RP) - - - R - - - - - - - - - - - Pattern : T Responds to P Scope : Between L and R Formula: $[](($L $\wedge(\neg R) \wedge \diamond R) \rightarrow (P \rightarrow (\neg RU(T \wedge \neg R)))UR)$ P : (i == 5) L : (i == 2) T : (i == 5) R : (i == 9) Interval: Yes Expected Result: Violation Actual Result: No violation

Test 20 - - L - - (RP) - - - - - - - - - - - - - - - Pattern : T Responds to P Scope : After L Formula: $[](\mathrm{L} \rightarrow [](P \rightarrow \diamond T))$ P : (i == 5) T : (i == 5) L : (i == 2) Interval: Yes Expected Result: Violation Actual Result: No violation	Test 36 - - L - - (RP) - - - R - - - - - - - - - - - Pattern : T Responds to P Scope : Between L and R Formula: $[]((\mathrm{L} \wedge(\neg R) \wedge \diamond R) \rightarrow (P \rightarrow (\neg RU(T \wedge \neg R)))UR)$ P : (i == 5) L : (i == 2) T : (i == 5) R : (i == 9) Interval: Yes Expected Result: Violation Actual Result: No violation
Test 43 - - L - - (RP) - - - R - - - - - - - - - - - Pattern : T Responds to P Scope : After L Until R Formula: $[]((\mathrm{L} \wedge(\neg R)) \rightarrow (P \rightarrow (\neg RU(T \wedge \neg R)))WR)$ P : (i == 5) L : (i == 2) T : (i == 5) R : (i == 9) Interval: Yes Expected Result: Violation Actual Result: No violation	Test 44 - - L - - (RP) - - - - - - - - - - - - - - R Pattern : T Responds to P Scope : After L Until R Formula: $[]((\mathrm{L} \wedge(\neg R)) \rightarrow (P \rightarrow (\neg RU(T \wedge \neg R)))WR)$ P : (i == 5) L : (i == 2) T : (i == 5) R : (i == limit) Interval: Yes Expected Result: Violation Actual Result: No violation

Fig. 7. Examples of SPS Failed Tests

Generic Verification of Security Protocols

Abdul Sahid Khan, Madhavan Mukund, and S. P. Suresh

Chennai Mathematical Institute,
92 G.N. Chetty Road, T.Nagar, Chennai 600 017, India
{sahid, madhavan, spsuresh}@cmi.ac.in

Abstract. Security protocols are notoriously difficult to debug. One approach to the automatic verification of security protocols with a bounded set of agents uses logic programming with analysis and synthesis rules to describe how the attacker gains information and constructs new messages.

We propose a generic approach to verifying security protocols in SPIN. The dynamic process creation mechanism of SPIN is used to nondeterministically create different combinations of role instantiations. We incorporate the synthesis and analysis features of the logic programming approach to describe how the intruder learns information and replays it back into the system. We formulate a generic "loss of secrecy" property that is flagged whenever the intruder learns private information from an intercepted message. We also describe a simplification of the Dolev-Yao attacker model that suffices to analyze secrecy properties.

1 Introduction

1.1 Background

Security protocols are specifications of communication patterns which are intended to let agents share secrets over a public network. They are required to perform correctly even in the presence of malicious intruders who listen to the message exchanges that happen over the network and also manipulate the system (by blocking or forging messages, for instance). Obvious correctness requirements include secrecy: an intruder cannot read the contents of a message intended for others, and authenticity: if B receives a message that appears to be from agent A and intended for B, then A indeed sent the same message intended for B in the recent past.

The presence of intruders necessitates the use of encrypted communication. A wide variety of cryptographic primitives – some of whose development involves quite sophisticated number theory – is an essential part of the protocol designer's toolkit. But it has been widely acknowledged that even the use of the most perfect cryptographic tools does not always ensure the desired security goals. (See [AN95] for an illuminating account.) This situation arises primarily because of logical flaws in the design of protocols.

Quite often, protocols are designed with features like ease of use, efficiency etc. in mind, in addition to some notion of security. For instance, if every message

P. Godefroid (Ed.): SPIN 2005, LNCS 3639, pp. 221–235, 2005.

of a protocol were signed in the sender's name and then encrypted with the receiver's public key, it appears as if a lot of the known security flaws do not occur. But it is not usual for every message of a protocol to be signed. This could either be for reasons of efficiency or because frequent use of certain long-term keys might increase the chance of their being broken using cryptanalysis. Great care needs to be exercised in such situations. The following example protocol highlights some of the important issues nicely. It is based on a protocol designed by Needham and Schroeder [NS78] and is aimed at allowing two agents A and B to exchange two independent, secret numbers. It uses public-key encryption but does not require agents to sign their messages.

$$\begin{array}{l}\text{Msg 1. } A \rightarrow B : \{x, A\}_{pubk_B} \\ \text{Msg 2. } B \rightarrow A : \{x, y\}_{pubk_A} \\ \text{Msg 3. } A \rightarrow B : \{y\}_{pubk_B}\end{array}$$

Here $pubk_A$ and $pubk_B$ are the public keys of A and B, respectively, and $\{x\}_k$ is the notation used to denote x encrypted using key k. In the protocol, x and y are assumed to be newly generated, unguessable (with high probability, of course!), previously unused numbers, also called nonces (nonce stands for "*n*umber *once* used"). In message 2, B includes A's nonce. On seeing it A is assured that B has received message 1, since only B can decrypt the first message and use x in a later message. Similarly on receipt of the third message, B is assured of A's receipt of y.

At the end of a session of the protocol, both A and B share the secrets x and y and both also know that the other agent knows x and y. But it has been shown [Low96] that x and y are not necessarily known *only to A and B*. (Such a property needs to be satisfied if we want to use a combination of x and y as a key shared between A and B, for example.) The attack (called Lowe's attack) is given below:

$$\begin{array}{lll}\text{Msg } \alpha.1. & A \rightarrow I & : \{x, A\}_{pubk_I} \\ \text{Msg } \beta.1. & (I)A \rightarrow B & : \{x, A\}_{pubk_B} \\ \text{Msg } \beta.2. & B \rightarrow (I)A & : \{x, y\}_{pubk_A} \\ \text{Msg } \alpha.2. & I \rightarrow A & : \{x, y\}_{pubk_A} \\ \text{Msg } \alpha.3. & A \rightarrow I & : \{y\}_{pubk_I} \\ \text{Msg } \beta.3. & (I)A \rightarrow B & : \{y\}_{pubk_B}\end{array}$$

In the above attack, $(I)A \rightarrow B : m$ means that the intruder is sending message m to B in A's name, whereas $A \rightarrow (I)B : m$ means that the intruder is blocking a message sent by A intended for B. The above attack consists of two parallel sessions of the protocol, one (whose messages are labelled with α) involving A as the initiator and I as responder, and the other (whose messages are labelled with β) involving I (in A's name) as the initiator and B as the responder. (This shows that the names A, B, x and y mentioned in the protocol specification are just placeholders or abstract names, which can be concretely instantiated in different ways when the protocol is run. So according to A and B, they have just had a normal protocol session with I and A, respectively. But I knows better!) After

the fifth message above, the intruder gets to know y which is the secret generated by B in a session with someone whom B believes to be A. This shows that the protocol does not satisfy the following property: *whenever an agent B engages in a session of the protocol as a responder and B believes that the initiator is A, then the secret generated by B is known only to A and B.* The seriousness of this flaw depends on the kinds of use the protocol is put to. It is worth noting that this attack does not depend on weaknesses of the underlying encryption mechanism (nor even on some keys being guessed by chance). It is also worth noting that this attack on the (simple enough) Needham-Schroeder protocol was discovered seventeen years after the original protocol was proposed. [Low96] also suggests a fix for the protocol:

$$\begin{array}{l}\text{Msg 1. } A \rightarrow B : \{x, A\}_{pubk_B} \\ \text{Msg 2. } B \rightarrow A : \{x, y, B\}_{pubk_A} \\ \text{Msg 3. } A \rightarrow B : \{y\}_{pubk_B}\end{array}$$

It is easy to see that the above attack does not happen anymore, but that still doesn't prove that the protocol does not have any vulnerabilities.

The above discussion illustrates the pitfalls in security protocol design, and also highlights the need for a systematic approach to protocol design and analysis. There are two possible approaches:

- Development of a design methodology following which we can always generate provably correct protocols. The work in [AN96], which gives a flavour of the kinds of useful heuristics which improve protocol design, is a step in this direction.
- Development of systematic means of analysing protocols for possible design flaws. The bulk of the work in formal methods for security protocols focuses on this approach. Here again, there are two possibilities:
  - Development of methods for proving the correctness of certain aspects of protocols.
  - Development of systematic methods for finding flaws of those protocols which are actually flawed.

There has been much work in applying *automated theorem proving* ([Pau98] and [Bol97] are some representative papers) and specialised *belief logics* ([BAN90], [AT91], [GNY90], [SC01] is a sampling of the literature) to prove properties of protocols.

In this paper, we describe our experiments with applying the SPIN model-checking tool to verify security protocols. The outline of the paper is as follows. In the rest of this section, we discuss the issues which arise in model checking security protocols, and also set our work in the context of other research in this area. In Section 2, we develop a formal model for security protocols. In the next section, we give a high-level description of how we verify security protocols in Spin, using the Needham-Schroeder protocol as a running example. Our approach makes use of some new observations about the properties required of the intruder. We formally justify our assumptions in Section 4. The final section

summarizes the work and discusses future directions. The full Promela code for modelling the Needham-Schroeder protocol is provided in the technical report [KMS05].

1.2 Model Checking Security Protocols

Much of the literature in formal methods for security protocols is devoted to methods for detecting flaws in protocols using the *model checking* approach – [Low96], [LR97], [MMS97], [Sch96], and [Sch97] is a representative sample. This approach has enjoyed great success in unearthing bugs in many protocols – long after the protocols had been put into use, in some cases. [CJ97] is a good reference for the many attacks which have been uncovered by formal verification tools.

Model checking security protocols is particular challenging. Unlike many other communication protocols, the actual data which is transferred in the message exchanges is also of importance in security protocols. For instance, in the Needham-Schroeder protocol, the nonce which B receives in the third message should be the same as the one it sent in the second message. But that is not all. The names A, B, n which occur in the protocol descriptions are placeholders. During a run of the protocol, they will be instantiated with concrete agent names like `sahid`, `madhavan`, `spsuresh`, etc., and concrete nonces – like a random 128-bit string, for instance. Further there may be many instantiations of the same roles occurring in one run of a protocol.

Technically speaking, we are dealing with infinite state systems and even simple problems like reachability are undecidable in various general settings. (See [DLMS99], [ALV02], [Sur03], among others, for more details.) Therefore we impose various external bounds, mainly on the number of different sessions or plays (instantiations of roles) that occur in each run of the protocol. But even then, there is a nontrivial amount of data manipulation involved, and the state space can be huge even when we consider a small number of sessions in each run. We attempt to master this complexity by looking at a generic secrecy property, and by observing the intruder structure can be considerably simplified when we search for a violation of this property.

Spin has been successfully used in model checking security protocols [MS02]. There are also reports of the use of other general purpose model checkers like Murφ to verify security protocols [MMS97]. Many attacks have been successfully discovered and rediscovered by these tools. But the reports in the literature either do not elaborate much on the details of the intruder model, or present an intruder which is designed with a particular attack in mind. Our work is characterised by a very *simple, yet general* intruder model, which helps us discover all breaches of a particular simple kind of *generic* secrecy property efficiently.

In particular, we allow our intruder to listen in on all the messages communicated over all the channels in the network, construct arbitrary messages out of learnt messages (using the message analysis and synthesis rules which are described in a later section), and at any point of time, send a constructed message to the appropriate agent under anyone's identity. But unlike the general Dolev-

Yao intruder, we do not allow our intruder to decrypt messages seen earlier using keys learnt later. This considerably simplifies the Promela code for the intruder.

There has been a lot of work based on logic programming that has been reported in the literature ([BP03], [MS01] and [DMTY97], for instance). To each protocol specification is associated a set of Horn clauses whose variables correspond to the variables occurring in the protocol specification. The negation of the desired security property is encoded as a goal formula, and one attempts to derive the formula using the clauses. In the course of a derivation, the variables are instantiated with concrete terms. These roughly correspond to spawning different sessions of the roles of the protocols, with different instantiations for the nonces and other secrets. It is proved that there is a proof of the goal if and only if there is a run of the protocol which constitutes a breach of the security property.

We feel that the model checking approach offers two advantages over the logic programming approach. Firstly, in the logic programming approach, extracting the counterexample run from a proof of the goal is not always a straightforward matter, whereas the SPIN model checker readily provides us with the counterexample run when it detects a violation of the desired property. Secondly, if the model checker doesn't report an error, we know that there is no attack on the protocol within the bounds set on the parameters. On the other hand, with the logic programming approach, there is no easy way of directly bounding the number of sessions in each run of the protocol.

2 A Formal Model for Security Potocols

In this section we present a formal model for security protocols, which includes a description of the Dolev-Yao intruder [DY83]. Our presentation is necessarily brief. A more detailed presentation can be found in [RS05].

We start with a (potentially infinite) set of *agents* Ag, which includes the *intruder* I and the others, who are called *honest agents*. We also start with a set of *keys* K which includes long-term public, private, and shared keys, as well as temporary session keys. For every key k, we denote by $\overline{k}$ its *inverse*. Public keys and their corresponding private keys are inverses of each other, while shared keys are their own inverses. We also assume an initial distribution of long-term keys (for example, A has his private key, everyone's public key, and a shared key with everyone else) which is common knowledge. The set of keys known to A initially is denoted K_A. We also assume a countable set of *nonces* N. $\mathcal{T}_0$, the set of *basic terms*, is defined to be $K \cup N \cup Ag$. The set of *information terms* is defined to be

$$\mathcal{T} \quad ::= \quad m \mid (t_1, t_2) \mid \{t\}_k$$

where m ranges over $\mathcal{T}_0$ and k ranges over K. These are the terms used in the message exchanges below. We use the standard notion of subterms of a term to define $ST(t)$ for every term t.

We model communication between agents by *actions*. An action is either a *send action* of the form $A!B{:}(M)t$ or a *receive action* of the form $A?B{:}t$. Here A

and B are distinct agents, A is honest; and M denotes the set of nonces and keys occurring in t which have been freshly generated during this (send) action. We define $term(A!B{:}(M)t)$ and $term(A?B{:}t)$ to be t. The agent B is (merely) the intended receiver in $A!B{:}(M)t$ and the purported sender in $A?B{:}t$. As we will see later, every send action is an instantaneous receive by the intruder, and similarly, every receive action is an instantaneous send by the intruder.

Definition 1. *A* **protocol** *is a pair* $\mathsf{Pr} = (\mathsf{C}, \mathsf{R})$ *where* $\mathsf{C} \subseteq \mathcal{T}_0$ *is the set of* constants *of* Pr *(intended to have a fixed interpretation in all runs of* Pr*, unlike fresh nonces and keys); and* R*, the set of* roles *of* Pr*, is a finite nonempty subset of* Ac^+ *each of whose elements is a sequence of* A*-actions for some honest agent* A*.*

The semantics of a protocol is given by the set of all its runs. A run is got by instantiating each role of the protocol in an appropriate manner, and forming admissible interleavings of such instantiations. We present the relevant definitions below.

An *information state* s is a tuple $(s_A)_{A \in Ag}$ where $s_A \subseteq \mathcal{T}$ for each agent A. $\mathcal{S}$ denotes the set of all information states. Given a protocol $\mathsf{Pr} = (\mathsf{C}, \mathsf{R})$, $\mathsf{init}(\mathsf{Pr})$, the *initial state of* Pr is defined to be $(\mathsf{C} \cup K_A)_{A \in Ag}$.

A *substitution* σ is a partial map from $\mathcal{T}_0$ to $\mathcal{T}$ such that for all $A \in Ag$, if $\sigma(A)$ is defined then it belongs to Ag, for all $n \in N$, if $\sigma(n)$ is defined then it belongs to N, and for all $k \in K$, if $\sigma(k)$ is defined then it belongs to K. The definition is generalised to arbitrary terms and actions in the usual manner.

The notion of information state that we use is very rudimentary. In general, a *control state* would include more detail like the number of current sessions each agent is involved in, how far it has progressed in each of them, and so on. For technical ease, we code up these details in the form of *events*. An event of a protocol Pr is a triple (η, σ, lp) such that η is a role of Pr, σ is a substitution, and $1 \leq lp \leq |\eta|$. The set of all events of Pr is denoted $Events(\mathsf{Pr})$. For an event $e = (\eta, \sigma, lp)$ with $\eta = a_1 \cdots a_\ell$, $act(e) \stackrel{\text{def}}{=} \sigma(a_{lp})$ and $term(e) = term(act(e))$. If $lp < |\eta|$ then $(\eta, \sigma, lp) \rightarrow_\ell (\eta, \sigma, lp + 1)$. For any event e, $LP(e)$, the *local past* of e, is defined to be the set of all events e' such that $e' \stackrel{+}{\rightarrow}_\ell e$.

We intend a run of a protocol to be an admissible sequence of events. A very important ingredient of the admissibility criterion is the enabling of events given a particular information state. To treat this formally, we need to define how the agents (particularly the intruder) can build new messages from old. This is formalised by the notion synth and analz derivations.

Definition 2. *A* sequent *is of the form* $T \vdash t$ *where* $T \subseteq \mathcal{T}$ *and* $t \in \mathcal{T}$*.*

An analz*-proof (*synth*-proof)* π *of* $T \vdash t$ *is an inverted tree whose nodes are labelled by sequents and connected by one of the* analz*-rules (*synth*-rules) in Figure 1, whose root is labelled* $T \vdash t$*, and whose leaves are labelled by instances of the* Ax_a *rule (*Ax_s *rule). For a set of terms* T*,* $\mathsf{analz}(T)$ *(*$\mathsf{synth}(T)$*) is the set of terms* t *such that there is an* analz*-proof (*synth*-proof) of* $T \vdash t$*. For ease of notation,* $\mathsf{synth}(\mathsf{analz}(T))$ *is denoted by* $\overline{T}$*.*

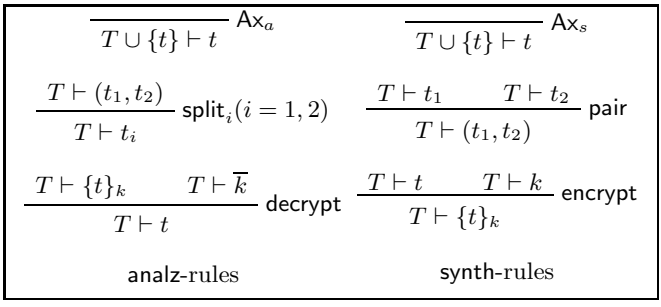


Fig. 1. analz and synth rules

Definition 3. *The notions of an action enabled at a state and update of a state on an action are defined as follows:*

- $A!B{:}(M)t$ *is* enabled *at* s *iff* $t \in \overline{s_A \cup M}$.
- $A?B{:}t$ *is* enabled *at* s *iff* $t \in \overline{s_I}$.
- $update(s, A!B{:}(M)t) \stackrel{\text{def}}{=} s'$ *where* $s'_A = s_A \cup M$, $s'_I = s_I \cup \{t\}$, *and for all agents* C *distinct from* A *and* I, $s'_C = s_C$.
- $update(s, A?B{:}t) \stackrel{\text{def}}{=} s'$ *where* $s'_A = s_A \cup \{t\}$ *and for all agents* C *distinct from* A, $s'_C = s_C$.

Definition 4. *Given a protocol* Pr *and a sequence* $\xi = e_1 \cdots e_k$ *of events of* Pr, *infstate*(ξ) *is defined to be update*(init(Pr), $act(e_1) \cdots act(e_k)$). *An event* e *is said to be* enabled at ξ *iff* $LP(e) \subseteq \{e_1, \ldots, e_k\}$ *and* $act(e)$ *is enabled at infstate*(ξ).

Definition 5. *Given a protocol* Pr, *a sequence* $\xi = e_1 \cdots e_k$ *of events of* Pr *is said to be a* run *of* Pr *iff:*

- *for all* $i : 1 \leq i \leq k$, e_i *is enabled at* $e_1 \cdots e_{i-1}$,
- *for all* $i : 1 \leq i \leq k$, $NT(e_i) \cap ST(\mathsf{init}(\mathsf{Pr})) = \emptyset$, *and for all* $i < j \leq k$, $NT(e_i) \cap NT(e_j) = \emptyset$. *(This is the* unique origination property *of runs.)*

We denote the set of runs of Pr *by* $\mathcal{R}(\mathsf{Pr})$.

In using the model checker, we typically consider runs of Pr which involve a bounded number of instantiations of roles.

Definition 6. *An atomic term* m *is* secret *at a state* s *if* $m \in \mathsf{analz}(s_A) \setminus \mathsf{analz}(s_I)$ *for some* $A \in Ag$ *– it is known to some honest agent but not to the intruder. Given a protocol* Pr, *and a run* ξ *of* Pr, m *is secret at* ξ *if it is secret at infstate*(ξ). *A run is said to be* leaky *if some atomic term* m *is secret at a prefix of* ξ *but not secret at* ξ.

The secrecy problem *is the problem of verifying whether a given protocol* Pr *has a leaky run.*

3 Protocol Verification Using Spin

In this section, we discuss our Promela modelling of the Needham-Schroeder protocol, as a means of illustrating our approach. The full code can be found in the technical report [KMS05]. The general approach to verifying security protocols in model-checking tools is by now standard, and amounts to two major steps.

- Formalize the protocol.
- Formalize the behaviour of the intruder.

Our aim is to propose a methodology to achieve these two steps in a manner that can be automated, given a reasonable description of the protocol.

3.1 Formalization of the Protocol

As we have seen, a protocol can be described as a pattern of messages exchanged between participants playing specified roles. Each role is easily described as a `proctype` in Spin. What is difficult to formalize is the choice in the way these roles are instantiated in order to find flaws in the protocol.

In our approach, each role instantiation corresponds to a fresh instance of the given `proctype`. To account for the fact that the same agent may play multiple roles, when a `proctype` is instantiated we also provide it with an integer identity. The intruder has a fixed identity, 1.

In the `init` process, we construct at least one instance of each proctype. We then, nondeterministically, construct multiple instances of proctypes to model arbitrary configurations. For instance, in our model of the Needham-Schroeder protocol, `procI`, `procA` and `procB` are the proctypes for the roles intruder, role A and role B, respectively. The code in Figure 2 constructs one instance of each proctype and then upto `KEY_MAX` more instances, each of which is either `procA` or `procB`.

We also have to model nonces. Rather than deal with them symbolically, as in done, for example, in [MS02], we use a shared global integer `used_nonce`. Whenever a process requires a nonce, it increments this global variable and uses the corresponding value. Since protocols are usually very short and we only use a limited number of instantiations of each role, we can safely define `used_nonce` as `byte` without running out of fresh nonces.

Public keys are implicitly identified with the identity of the agent. Thus, the public key of agent i is just i. Similarly, shared keys can be encoded using a pair of agent identities.

We use an array of `proc_chan` of synchronous Spin channels to model the actual channels in the system. Each instance of a role with identity i reads messages on channel `proc_chan[i]`. As we shall see, we allow the intruder to read messages on every channel `proc_chan[i]`. We could also use a model in which messages are always routed via the intruder, in which case we need to include the identity of the recipient in each message.

```
init {
  byte j = 3;

  run procI(1);  run procA(2);  run procB(3);

  do  /* create more processes nondeterministically */
    :: break;
    :: j++;
       if
         :: (j >= KEY_MAX) -> break;
         :: else -> if
                      :: run procA(j);
                      :: run procB(j);
                    fi;
       fi;
  od;
}
```

Fig. 2. Nondeterministic instantiation of roles

3.2 Formalization of the Intruder

The Dolev-Yao intruder model can be modelled by a process that repeatedly performs the following steps:

- Nondeterministically intercept a message on some channel and update its information.
- Nondeterministically generate a message on some channel using known information.

The intruder updates its information using analz rules and generates fresh messages using synth rules. In general, analz rules involve breaking up a message into its constituent parts, storing all the parts and decrypting previously stored messages using newly acquired keys. In the context of the secrecy problem described in the previous section, it suffices to use a simplified version of the Dolev-Yao intruder model in which the intruder never needs to use newly learned keys to decrypt previously stored messages. In effect, the analz phase consists of just breaking up the current message into its constituent parts and decrypting any encrypted component for which the intruder already possesses the decryption key.

Recall that we model nonces and keys using integers. We can thus model the information that the intruder knows using a boolean array indexed by integers (or pairs of integers). Whenever the intruder intercepts a message, the analz rules determine how these arrays are updated.

We also need to record stored messages. One approach would be to have a boolean array indexed by all combinations of message contents, where an entry is `true` whenever the corresponding message has been seen by the intruder. However, since we are looking at a limited number of interleavings of relatively

short sequences of messages, it is more efficient to just maintain a list of stored messages in an array.

Generating a message amounts to nondeterministically choosing a recipient, a message type and populating each field in the corresponding message with some known information of the appropriate type. Alternatively, the intruder could simply replay an entire stored message.

3.3 Formalization of the Secrecy Property

Recall that a secret is formally defined as information that is introduced during the run of the protocol by an honest agent but which is not known to the intruder at the time of its introduction. A secret leaks if it is intercepted by the intruder after having been sent by its originator to some other honest agent in the system.

We could also consider situations in which the secret leaks directly to the intruder. For instance, A could generate a nonce n_A that is sent unencrypted and is intercepted by the intruder. We do not consider this to be an unintended leakage of a secret. It is not difficult to modify our approach to consider such situations also as leakage of secrets.

Since we keep track of the information that the intruder knows, it is quite straightforward to flag an error when the intruder learns new information. To ensure that this meets our definition of when a secret leaks, for each secret nonce (respectively, key) i, we set the boolean `nonce_introduced[i]` (respectively, `key_introduced[i]`) to `true` when i is first received by any agent. In the Needham-Schroeder protocol, we are only interested in loss of secrecy for nonces, so we have the following code to flag loss of secrecy. This code is invoked whenever the intruder sets `known_nonce[k]` to `true`.

```
inline modifyFlawStatus(k) {
  if
    :: (nonce_introduced[k] == true) ->
       flaw = true;
    :: else -> skip;
  fi;
}
```

We can then write a generic verification condition for loss of secrecy in LTL as `[](!flaw)`.

3.4 Some Experimental Results

When we ran our Promela code for the Needham-Schroeder protocol through Spin, it discovered Lowe's attack, as shown in Figure 3. Notice that this counterexample only uses the basic 3 processes created initially, even though the Promela code permits the creation of additional role instantiations.

We also ran a modified version of this code in which the system necessarily generated 3 instantiations each of roles A and B. Interestingly, the counterexample reported by Spin corresponds to Lowe's attack involving the intruder and

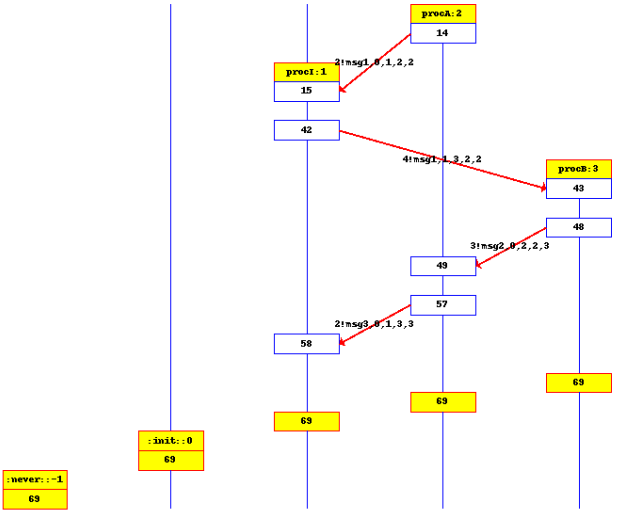


Fig. 3. Lowe's attack on the Needham-Schroeder protocol, discovered by SPIN

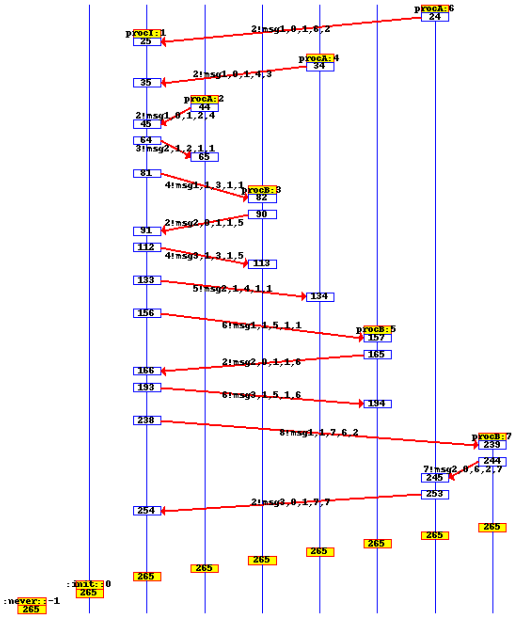


Fig. 4. Lowe's attack with 6 processes

the last two copies of A and B, with some irrelevant intervening messages between the other instances of A and B, as shown in Figure 4. When SPIN was asked to find the shortest counterexample, it found the version of Lowe's attack involving just the first three processes.

3.5 Comparison with earlier approaches

Ours is not the first attempt to use SPIN to verify protocols. Another attempt is described in [MS02]. In the earlier approach, instead of using synth and analz rules to describe the behaviour of the intruder, the intruder is allowed to generate arbitrary messages. Static analysis is used to limit the intruder's choices to "useful" messages. Nonces and other data are handled using symbolic names. Finally, the security property is formulated explicitly, keeping in mind the nature of the protocol. Though the authors claim that their approach can be automated, all of these factors appear to indicate the need for manual analysis before invoking SPIN.

In contrast, our approach formulates the intruder and the security property in a sufficiently generic manner that the Promela code for verifying a protocol can, in principle, be synthesized automatically from a suitable description of the protocol, using a system such as CAPSL [DMR00] or Casper [Low98].

4 A Simpler Intruder Model for Secrecy Properties

We now formalize the simplified intruder used in our Promela model and prove that it is as powerful as the Dolev-Yao intruder, as far as violating the secrecy property of Definition 6 is concerned.

In what follows, for ease of notation, we fix a protocol Pr which has a leaky run, and fix a leaky run $\xi = e_1 \cdots e_k$ of Pr of shortest length. For all $i : 1 \leq i \leq k$, we let t_i, ξ_i and T_i denote $term(e_i)$, $e_1 \cdots e_i$ and $(infstate(\xi_i))_I$, respectively. We let T_0 denote $\mathsf{C} \cup K_I$. For all $i \leq k$, define T'_i and N_i by induction as follows: $T'_0 = N_0 = \mathsf{C} \cup K_I$, $T'_{i+1} = T'_i \cup \mathsf{analz}(\{t_{i+1}\} \cup N_i)$ and $N_{i+1} = T'_{i+1} \cap \mathcal{T}_0$. In other words, after every event e_i, the intruder stores the nonces and keys known till now in N_i. As soon as e_{i+1} is seen, the keys from N_i are used to decrypt t_{i+1} and learn more nonces and keys.

Lemma 7. *There is some atomic term n_0 which is secret at ξ_{k-1} and which belongs to N_k. Further, for all $i \leq k$, if e_i is a receive event then $t_i \in \mathsf{synth}(T'_i)$.*

Proof. Since ξ is a leaky run of shortest length, none of its proper prefixes is a leaky run and hence it is clear that for all $i < k$, ξ_i is not leaky, which means that none of the terms known to the intruder at the end of ξ_i is a secret at ξ_j for any $j < i$. In other words, for all $i : 1 \leq i < k$ and for all $m \in \mathsf{analz}(T_i) \setminus \mathsf{analz}(T_{i-1})$, $m \not\in \mathsf{C}$, $m \not\in K_A$ for any $A \in Ag$, and $m \not\in ST(t_j)$ for any $j < i$.

Let m_0 be a secret at ξ_{k-1} which belongs to $\mathsf{analz}(T_k)$ (and, of course, does not belong to $\mathsf{analz}(T_{k-1})$). Let π be an analz-proof of $T_k \vdash m_0$, such that for all subproofs π_1 of π, if $T_k \vdash t$ labels the root of π_1 and $t \in \mathsf{analz}(T_i) \setminus \mathsf{analz}(T_{i-1})$ for some $i \leq k$, then all the leaves of π_1 are labelled only by terms t_j for $j \leq i$. Let ϖ be a subproof of π whose root is labelled $T_i \vdash n_0$ for some n_0 which is secret at ξ_{k-1}, and such that none of the terms labelling the nonroot nodes of ϖ is a secret at ξ_{k-1}.

We now prove by induction on the structure of ϖ the claim that for all terms t labelling a node of ϖ, $t \in T'_i$ for the least i such that $t \in \mathsf{analz}(T_i)$. Once we prove this claim, it immediately follows that $n_0 \in T'_k$ and, since $n_0 \in \mathcal{T}_0$, $n_0 \in N_k$. Further let e_i be a receive event for some $i \leq k$. Then from the admissibility conditions it is clear that $t_i \in \overline{T_{i-1}} = \mathsf{synth}(\mathsf{analz}(T_{i-1}))$. But it is an easy consequence of the claim that $\mathsf{analz}(T_j) \subseteq T'_j$ for all $j < k$, and it immediately follows from this that $t_i \in \mathsf{synth}(T'_{i-1})$.

We now turn to the proof of the claim. There are three cases to consider.

- The base case is when t labels a leaf node of ϖ. But this means that there is some $i \leq k$ such that $t = t_i$. By our assumption on π, it is clear that $t \not\in \mathsf{analz}(T_{i-1})$. Therefore i is the least number such that $t \in \mathsf{analz}(T_i)$. Clearly enough, $t \in T'_i$ as well.
- The other case is when t labels the conclusion of a split rule. Let i be the least number such that $t \in \mathsf{analz}(T_i)$. Let t' label the premise of the rule. Let i' be the least number such that $t' \in \mathsf{analz}(T_{i'})$. By our assumptions on π, it is clear that $i' \leq i$. By the induction hypothesis $t' \in T'_{i'} \subseteq T'_i$. From this another application of the split rule tells us that $t \in T'_i$.
- The most interesting case is when t labels a conclusion of a decrypt rule. Let i be the least number such that $t \in \mathsf{analz}(T_i)$. Clearly $\{t\}_k$ and $\overline{k}$ label the premises of the rule, for some key k. Let i' and i'' be the least numbers such that $\{t\}_k \in \mathsf{analz}(T_{i'})$ and $\overline{k} \in \mathsf{analz}(T_{i''})$. By our assumptions on π, it is clear that $i' \leq i$ and $i'' \leq i$.
  Now it cannot be the case that $i' < i''$. This is because, letting j be the least number such that k occurs as a subterm of t_j, it is easy to see that $j \leq i'$, and $\{k, \overline{k}\} \subseteq \mathsf{analz}((\mathit{infstate}(\xi_j))_A)$ for some $A \in \mathit{Ho}$. Now, since i'' is the least number such that $\overline{k} \in \mathsf{analz}(T_{i''})$, this means that $\overline{k}$ is a secret at ξ_j, which is a contradiction because it labels a nonroot node of ϖ. Therefore $i'' \leq i'$.
  By the induction hypothesis $\{t\}_k \in T'_{i'}$ and $\overline{k} \in T'_{i''}$. If $i'' < i'$, then $\overline{k}$ would belong to $T'_{i'-1}$ and hence to $N_{i'-1}$ (since $\overline{k} \in \mathcal{T}_0$). Then it is easy to see that $t \in T'_{i'} \subseteq T'_i$, as desired. On the other hand, if $i' = i''$ then $\{\{t\}_k, \overline{k}\} \subseteq \mathsf{analz}(\{t_{i'}\} \cup N_{i'-1})$, and hence $t \in \mathsf{analz}(\{t_{i'}\} \cup N_{i'-1}) \subseteq T'_{i'} \subseteq T'_i$, as desired.

The above lemma shows that whenever a Dolev-Yao intruder captures a secret, so does an intruder that does not decrypt earlier messages using keys it has learnt later. This justifies the intruder model of Section 3.

5 Conclusion

We have described an approach for generic verification of secrecy properties of security protocols using Spin. For some protocols, correctness is described in terms of authentication rather than secrecy. We do not yet have a uniform method for describing authentication properities in our framework. We have also not yet embarked on the ambitious programme of writing a compiler from

a specification language such as CAPSL into SPIN to automatically generate verification models for arbitrary protocols.

References

[ALV02] Roberto M. Amadio, Denis Lugiez, and Vincent Vanackère. On the symbolic reduction of processes with cryptographic functions. *Theoretical Computer Science*, 290(1):695–740, 2002.

[AN95] Ross Anderson and Roger M. Needham. Programming Satan's computer. In *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 426–441, 1995.

[AN96] Martin Abadi and Roger M. Needham. Prudent engineering practices for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22:6–15, 1996.

[AT91] Martin Abadi and Mark Tuttle. A Semantics fo a Logic of Authentication. In *Proceedings of the 10th ACM Annual Symposium on Principles of Distributed Computing*, pages 201–216, Aug 1991.

[BAN90] Michael Burrows, Martin Abadi, and Roger M. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, Feb 1990.

[Bol97] Dominique Bolignano. Towards a mechanization of cryptographic protocol verification. In *Proceedings of CAV'97*, volume 1254 of *Lecture Notes in Computer Science*, pages 131–142, 1997.

[BP03] Bruno Blanchet and Andreas Podelski. Verification of Cryptographic Protocols: Tagging Enforces Termination. In Andrew D. Gordon, editor, *Proceedings of FoSSaCS'03*, volume 2620 of *Lecture Notes in Computer Science*, pages 136–152, 2003.

[CJ97] John Clark and Jeremy Jacob. A survey of authentication protocol literature. Electronic version available at `http://www.cs.york.ac.uk./~jac`, 1997.

[DLMS99] Nancy A. Durgin, Patrick D. Lincoln, John C. Mitchell, and Andre Scedrov. The undecidability of bounded security protocols. In *Proceedings of the Workshop on Formal Methods and Security Protocols (FMSP'99)*, 1999.

[DMR00] G. Denker, J. Millen, and H. Ruess. The CAPSL Integrated Protocol Environment. Technical Report SRI-CSL-2000-02, SRI, October 2000. Available at `http://www.csl.sri.com/users/millen/capsl`.

[DMTY97] Mourad Debbabi, Mohamed Mejri, Nadia Tawbi, and Imed Yahmadi. Formal automatic verification of authentication protocols. In *Proceedings of the First IEEE International Conference on Formal Engineering Methods (ICFEM97)*. IEEE Press, 1997.

[DY83] Danny Dolev and Andrew Yao. On the Security of public-key protocols. *IEEE Transactions on Information Theory*, 29:198–208, 1983.

[GNY90] Li Gong, Roger Needham, and Raphael Yahalom. Reasoning About Belief in Cryptographic Protocols. In Deborah Cooper and Teresa Lunt, editors, *Proceedings 1990 IEEE Symposium on Research in Security and Privacy*, pages 234–248. IEEE Computer Society, 1990.

[KMS05] Abdul Sahid Khan, Madhavan Mukund, and S.P. Suresh. Generic verification of security protocols. Technical report, CMI, May 2005. Electronic version available at `http://www.cmi.ac.in/~spsuresh`.

[Low96] Gavin Lowe. Breaking and fixing the Needham-Schroeder public key protocol using FDR. In *Proceedings of TACAS'96*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166, 1996.

[Low98] Gavin Lowe. Casper: A compiler for the analysis of security protocols. *Journal of computer security*, 6:53–84, 1998.

[LR97] Gavin Lowe and Bill Roscoe. Using CSP to detect errors in the TMN protocol. *IEEE Transactions of Software Engineering*, 23(10):659–669, 1997.

[MMS97] John C. Mitchell, Mark Mitchell, and Ulrich Stern. Automated analysis of cryptographic protocols using Murφ. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 141–153, 1997.

[MS01] Jonathan K. Millen and Vitaly Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. In *ACM Conference on Computer and Communications Security*, pages 166–175, 2001.

[MS02] P. Maggi and R. Sisto. Using SPIN to Verify Security Protocols. In *Proceedings of the 9th International SPIN Workshop on Model Checking of Software*, number 2318 in Lecture Notes in Computer Science, pages 187–204, 2002.

[NS78] Roger M. Needham and Michael D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Communications of the ACM*, 21(12):993–999, 1978.

[Pau98] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of computer security*, 6:85–128, 1998.

[RS05] R. Ramanujam and S.P.Suresh. Decidability of context-explicit security protocols. *Journal of Computer Security*, 13(1):135–165, 2005.

[SC01] Paul F. Syverson and Iliano Cervesato. The logic of authentication protocols. In Ricardo Focardi and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design*, volume 2171 of *Lecture Notes in Computer Science*, pages 63–106, 2001.

[Sch96] Steve Schneider. Security properties and CSP. In *Proceedings of the IEEE Computer Society Symposium on Security and Privacy*, 1996.

[Sch97] Steve Schneider. Verifying authentication protocols with CSP. In *Proceedings of the 10th IEEE Computer Security Foundations Workshop*, 1997.

[Sur03] S.P. Suresh. *Foundations of Security Protocol Analysis*. PhD thesis, The Institute of Mathematical Sciences, Chennai, India, November 2003. Madras University. Available at `http://www.cmi.ac.in/~spsuresh`.

Using SPIN and Eclipse for Optimized High-Level Modeling and Analysis of Computer Network Attack Models

Gerrit Rothmaier[1], Tobias Kneiphoff[2], and Heiko Krumm[3]

[1] Materna GmbH, Dortmund, Germany
gerrit.rothmaier@materna.de
[2] Bosch Rexroth AG, Witten, Germany
tobias@kneiphoff.com
[3] Universität Dortmund, Dortmund, Germany
krumm@cs.uni-dortmund.de

Abstract. Advanced attack sequences combine different kinds of steps (e.g. attacker, protocol, and administration steps) on multiple networked systems. We propose a SPIN based approach for formal modeling and analysis of such scenarios. Our approach is especially suited for scenarios were protocol and network level aspects matter simultaneously. Typical attack sequences and not yet considered variants can be automatically found. The development of scenario models is supported by a modeling framework and the use of the high-level process specification language cTLA. A compiler translates the high-level cTLA models to Promela. This allows the powerful model-checking tool SPIN to be employed for analysis. Through integration of the compiler and SPIN into the Eclipse platform both model development and analysis are facilitated.

1 Introduction

Since security became an issue in computing, the objective of automatically analyzing system models and thus completely revealing immanent vulnerabilities and potential attack patterns exists. This objective is very ambitious, as we had to learn early, and may be reachable only under certain restrictions. Mainly there are two reasons why attempts for automated security analysis fail in practice. First, the development of suitable models is very expensive, since the model design is error-prone and tedious even if performed by well-educated and well-experienced designers. Second, analysis runs tend to exceed given time and memory limitations, since the analysis procedures have a high algorithmic complexity. These problems, by the way, are not restricted to automated security analysis but are already well-known in the general field of automated verification. Nevertheless, because security analysis of computer networks has to reason about unknown vulnerabilities, malfunctions and attack effects in comparably large systems, the search space is more complex and the problems therefore occur in an increased form.

Thus currently, a more realistic but still ambitious objective is to concentrate on a narrower field of interest and to lower the grade of automation by some forms of user

P. Godefroid (Ed.): SPIN 2005, LNCS 3639, pp. 236 – 250, 2005.

guidance. One recognizes that formal modeling and analysis have a certain value, even if they are only employed for the representation and precise description of known attacks, since they can lead to a better understanding and insight into the phenomena and so probably indirectly contribute to future enhancements. In fact, the current status already provides some enhancements. Still, the costs of model development demand for the specialization to closer fields of interest and the analysis tool limitations demand for user guidance and restricted analysis scopes. Experience, however, showed that analysis runs which concentrate on a certain known and predefined class of attacks can find new unexpected variants.

Advanced network attacks combine different aspects like carefully crafted attack steps, normal protocol execution steps and administrator actions on different hosts instead of plain vulnerability exploitations [Ver04]. Network level aspects, e.g. topology and connectivity, and protocol level aspects have to be considered simultaneously. Existing approaches concentrate on either protocol or network level aspects (cf. section 2).

We resort to formal modeling and analysis techniques for the functional aspects of concurrent process systems. Considering the problems of formal analysis which result from the expensive model design and the limitations of automated analysis tools, we follow up a combined approach which is mainly based on two elements. First, the system verification tool *SPIN* [Hol03] is applied for automated analysis in order to profit from its powerful analysis procedures. Second, the development of models is supported by a high-level modeling framework which provides model architecture guidelines and re-usable model definition components. The framework is based on the process specification language *cTLA* [HK00]. *cTLA* is a variant of Leslie Lamport's *Temporal Logic of Actions TLA* [Lam94] and provides for the modular definition of process types and the derivation of new process types by refinement and composition. Therefore, *cTLA* facilitates the efficient re-use and adaptation of framework elements. In comparison to *SPIN's* model description language *Promela* more abstract and compositional model definitions are supported. The link to *SPIN* is provided by a compiler translating *cTLA* model definitions into *Promela*. Besides just translating models the compiler applies model optimizations. Moreover, the practical application of our approach is supported by means of a model development environment, which is implemented by extensions to the well-known software development tool *Eclipse* [Ecl05].

The approach has already been applied successfully to the modeling and analysis of different scenarios. [RPK04] presents the modeling and *SPIN*-based analysis of *ARP* spoofing like attacks respectively erroneous network management actions in a small LAN. Furthermore, we researched the modeling and analysis of *RIP* routing attacks.

This paper focuses on the compositional structure of our models, the optimized translation of our models to *Promela* and the integration of *SPIN* and related tools into the *Eclipse* universal tool platform. As a next step, after addressing related work, we give an outline of the modeling framework and the model definition language *cTLA*. Two example models clarify the framework's application and highlight the compositional structure of our *cTLA* modeling. Then we discuss the principles of translating *cTLA* models to *Promela*. Model optimizations are outlined in the next

section. Finally we describe how model editing, model translation, and *SPIN*-based analysis are integrated into *Eclipse*.

2 Related Work

Formal analysis and verification of security properties can be generally structured into *program and protocol verification*. Program verification shall enhance the trustworthiness of software systems (e.g. [BR00]). In protocol verification security weaknesses of protocols shall be found. Basic and cryptographic protocols (e.g. [MS02]) are particularly interesting. In both fields a variety of methods is applied, including classic logic and algebraic calculi (e.g. [KK03]), special calculi (e.g. [BAN89]), and process system modeling techniques (e.g. [LBL99]). Different kinds of analysis tools are used, including logic programming environments like *Prolog*, expert system shells, theorem provers, algebraic term rewriting systems, and especially model checkers. Some approaches even combine several analysis techniques [Mea96].

The formal modeling and analysis of complex, intertwined attack types in computer network scenarios is a relatively new field. Existing approaches either focus abstractly on protocols and disregard network level aspects like topology, connectivity, and routing or the other way around. For example, in [RS02] the analysis of attack sequences resulting from the combined behavior of system components is described for a *single* host. A process model is used which is specified in a *Prolog* variant. Security properties are expressed by labeling states safe and unsafe. Execution sequences which lead to unsafe states and correspond to vulnerability executions are searched using a *Prolog* based programming environment.

In [AR00, NBR02] an approach called *topological vulnerability analysis* is presented. A network of hosts is checked for attack sequences consisting of combining predefined vulnerabilities. The host modeling consists of two sets representing existing vulnerabilities and attacker access level. Network topology is modeled using a multi-valued connectivity matrix. Protocols are represented very simply through fixed values in the connectivity matrix; no sending, receiving, or processing of protocol elements is modeled. Exploit definitions have to be given with the model. Using *SMV* possible combinations of the given vulnerabilities leading to the violation of a property (e.g. attacker has root access level on a specified host) are analyzed.

Our approach supports modeling and analysis of network and protocol level aspects simultaneously in a single model. With respect to efficient modeling, the framework makes use of techniques invented for high-performance implementation of protocols. In particular we learned from the *activity thread* implementation model which schedules activities of different protocol layers in common sequential control threads [Svo89], and from integrated layer processing which combines operations of different layers [AP93]. *Partial order reductions*, proposed in [ABH97], have a strong relationship to the *partial order reduction* implementation model providing the basis for the elimination of nondeterministic execution sequences. Furthermore,

approaches for *Promela* level model optimizations have to be mentioned. Many interesting low-level optimizations are described in [Ruy01].

3 Modeling Framework

In order to foster re-use and reduce the effort needed for modeling, we looked into the possibility of creating a framework for formal modeling of computer networks. Because frameworks usually make heavy use of object-oriented mechanisms for composing elements and describing their relationships, we have to use a specification language that can express such concepts as well.

cTLA 2003 Specification Language

cTLA is based on *TLA* [Lam94], but supports explicit notions of process instances, process types, and process type composition [HK00]. Furthermore, *cTLA* 2003 adds object-oriented process composition types. Here we only give a conceptual overview of *cTLA* 2003. A language oriented description can be found in the technical report [RK03].

A *cTLA* specification describes a *state-transition system* which is composed of subsystems. These subsystems may be composed of further subsystems but are finally sets of process instances. Thus, after resolving subsystems, a *cTLA* system is always a composition of *process instances*.

Process instances belong to processes, which are typed. Each *process type* describes its own, self-contained state-transition system. In the simplest case, a process type does not use composition, but is completely self-contained. The state space of such a *simple process type* is completely defined through the set of local variables. Its transitions consist just of the process's actions. Actions are parameterized and describe atomic transitions consisting of guards and effects. *Guards* define conditions that must be met to make the action executable, *effects* describe the state changes triggered by the action's execution.

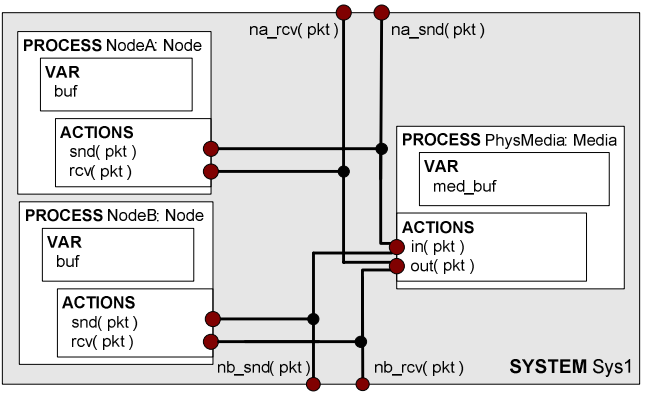


Fig. 1. Action Coupling in a Simple cTLA System

Object oriented mechanisms are introduced with the *cTLA process composition type* `EXTENDS` and `CONTAINS`. These composition types allow new process types to be derived from other process types. The state space and the transitions of such process types depend not only on the locally defined variables and actions, but also on the state and transitions of the inherited process types.

Synchronization and communication between process instances is done via *joint actions*. Joint actions couple 1..n actions of process instances, i.e. their guards and effects are conjugated. All process instances not taking part in a joint action perform a *stuttering step*. On the system level, *system actions* have to be given to define the possible state transitions.

As an example, consider **Figure 1**. A simple *cTLA* System `Sys1` contains three process instances `NodeA`, `NodeB` and `PhysMedia` which are instances of types `Node` respectively `Media`. Each instance has it's own variables and actions. Four system actions, `na_rcv(pkt)`, `na_snd(pkt)`, `nb_snd(pkt)` and `nb_rcv(pkt)` are defined through *coupling* of instances' actions. For example, action `na_rcv(pkt)` couples `NodeA`'s `rcv` action with `PhysMedia`'s out action and `NodeB` performs a stuttering step.

Computer Network Modeling Framework

Frameworks as known from the world of object oriented programming consist of classes and their relationships. With process types and process composition types we have similar mechanisms available in *cTLA* 2003. We aim to transfer qualities known from object oriented frameworks like "natural modeling", broad level re-use of proven elements and architectures to formal modeling, especially of computer network related scenarios. Thus, we developed a modeling framework for TCP/IP based computer networks in *cTLA* 2003. We only give an overview of the framework here, a structure diagram and element-by-element description can be retrieved via our web site [Rot04].

The framework is structured into layers horizontally. The first layer, *Enumerations & Functions*, is used to define the network topology, initial address assignment and protocols desired for a model. For example, the enumeration `ZoneIdT` contains the model's zones (usually matching Ethernet segments), the function `fSrcToIa` is used to assign initial addresses and the enumeration `ProtocolT` symbolically lists protocols required in the model.

The second layer, *Data Types*, contains common data types for interfaces, packets and buffers used by other elements of the framework. For instance, the type `InterfaceT` combines attributes of an interface; `PacketT` is a record used to represent a packet and `PacketBufT` defines a buffer for such a packet.

Finally, the third layer, *Process Types*, provides core process types. For example, process type `RouterIpNode` models the basic behavior of a forwarding IP node and `HostIpNode` represents a passive IP host node. Through inheritance behavior is specialized. For example, `ActiveHostIpNode` adds behavior for the processing and sending of packets to `HostIpNode`.

Usually, all layers of the framework collaborate to model a conception. For example, a scenario's network topology is modeled by several functions (e.g.

`fSrcToZone`) and enumerations (e.g. `ZoneIdT`), together with appropriate handling by processes (e.g. `Media`, `HostIpNode`, `RouterIpNode`) and their actions (e.g. `out`, `rcv`) which are parameterized with data types (e.g. `PacketT`).

During the design of the core process types we realized the usefulness of ideas known from *efficient protocol implementation techniques*. Especially the *activity thread* [Svo89] approach, which schedules activities of different protocol layers in common sequential control threads, and *integrated layer processing* [AP93], which combines operations of different layers, were helpful. Thus the number of concurrent execution paths (for packet processing) is smaller. This resembles *partial order reduction* techniques but is contained in framework derived models already. Fewer actions and buffers in the *cTLA* model lead to a reduced *SPIN* state-vector size and less possible transitions in the *Promela* model.

Example models use the basic structure and node types given by the framework. Of course, they usually have to add their own specific node types, e.g. `RipRouterIpNode`. These node types are derived from the framework's basic nodes and add data structures and behavior e.g. for processing additional protocols like RIP routing.

4 Example Models

Our approach has already been successfully used for the modeling and analysis of two example scenarios, the IP-ARP [RPK04] and RIP models. We focus on the structure of the models and the relationship to the framework here.

IP-ARP Model

In the IP-ARP scenario, a small LAN with three hosts running a basic TCP/IP stack is modeled. This scenario is analyzed for confidentiality violations, i.e. packets received by non-intended recipients.

The IP-ARP *cTLA* model structure is based on a preliminary version of the current framework. For the hosts, instances of the `IpArpNode` process type, which extends `HostIpNode` from the framework, are used. The `IpArpNode` process type adds support for a low-level ARP protocol layer. ARP queries are broadcasted for resolving yet unknown IP to hardware address mappings. ARP replies are processed and a local ARP cache is managed. On the IP level, changing of assigned IP addresses through management actions is added. Furthermore, packets to destination IP addresses with hardware addresses not yet in the ARP cache are buffered and the ARP layer is signaled. The LAN itself is modeled by a one zone `Media` instance with appropriate supporting enumerations (`ZoneIdT`, `NodeIdT`) and topology functions (`fSrcToZone`). Finally, the system is defined as an instance of the `IpArpExample` process type which is in turn a composition of one `Media` and three `IpArpNode` instances.

After translation, depending on the inserted assertions modeling confidentiality properties, various violating sequences can be found. Interestingly, these sequences can be triggered by both ARP attacks and certain IP change management actions.

RIP Model

The RIP scenario consists of three LANs, connected by three routers R1, R2, R3 in a triangle-like fashion. Representative hosts H1, H2, HA are chosen from the LANs. The hosts are TCP/IP nodes, the routers additionally run the RIP protocol. In this scenario, man-in-the-middle attacks through forged RIP updates by host HA on communication between hosts H1 and H2 are analyzed.

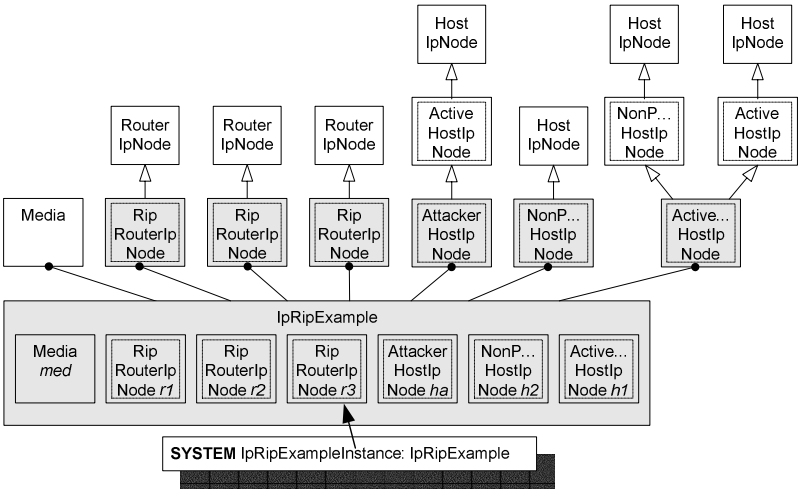


Fig. 2. Compositional Structure of the RIP Model

In the RIP model, all routers are instances of `RipRouterIpNode` (cf. **Figure 2**), which is based on `RouterIpNode` from the framework. The `RipRouterIpNode` type adds functionality for processing and sending RIP update messages and updating its routing table accordingly. The hosts are modeled by different process types ultimately based on `HostIpNode` according to their role in the scenario (attacker, active or passive communication partner). The LANs are modeled by a six zones `Media` instance with appropriate helper enumerations and functions. The system is an instance of the composed process type `IpRipExample` containing `Media`, three `RipRouterIpNode`, and three `HostIpNode` derived instances.

Again, depending on the exact property modeling, various attack sequences can be found. **Figure 3** shows an example sequence. The sequences resemble typical routing attack ideas mentioned in [BHE01].

5 Translating cTLA Specifications to Promela

To be able to leverage both a high level *cTLA* based framework and *SPIN*'s powerful capabilities for checking *Promela* specifications, we engineered the *cTLA2PC* tool. It takes a *cTLA* specification as input and transforms it to an equivalent, optimized *Promela* specification. Alternatively, the output of a simplified, "flat" *cTLA*

specification is possible as well. *cTLA2PC* is based on the ANTLR parser construction kit. In the following section, we give a short description of the most current version *cTLA2PC* version ("cTLA2PC 2").

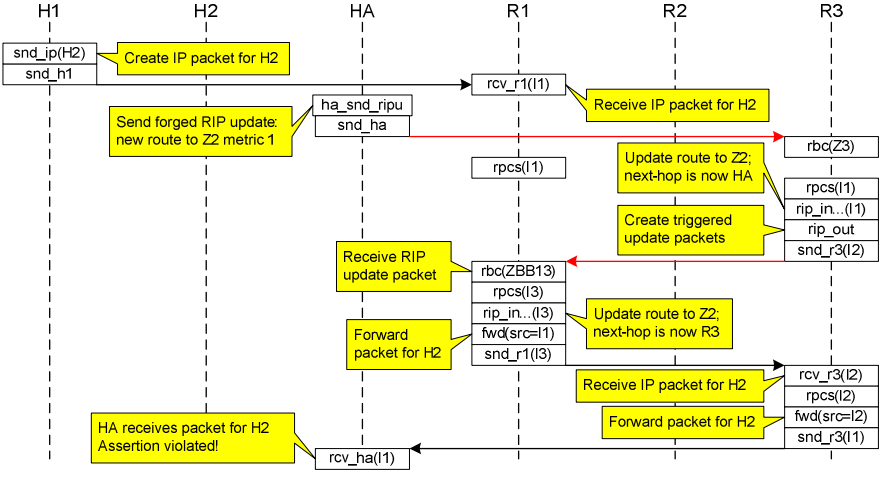


Fig. 3. Simplified Attack Sequence in the RIP scenario

The translation process starts with the scanner and parser components of *cTLA2PC*. If syntax errors are encountered, *cTLA2PC* prints an error message and the translation halts right after the parsing phase. After scanning and parsing, the semantic analysis is applied. Semantic analysis includes type checking of action parameters, function return values and assignments. Again, errors are flagged and stop the translation process.

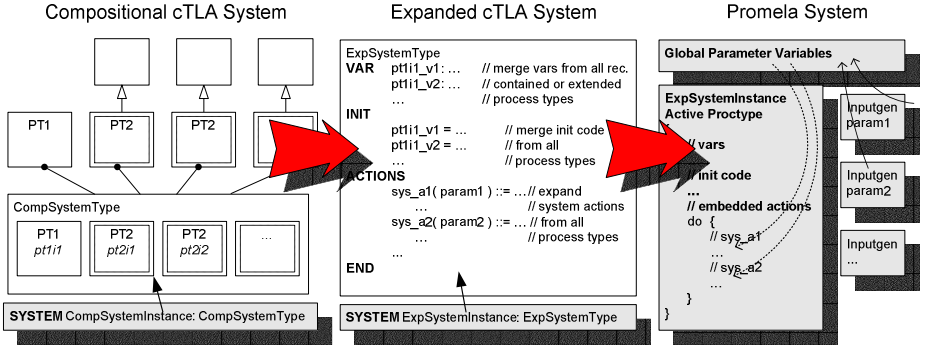


Fig. 4. Transforming a Compositional cTLA System to Promela

The key phase for the translation of *cTLA* specifications to *Promela* is the *expansion* (cf. **Figure 4**). It transforms a compositional *cTLA* system to an expanded

cTLA system. A compositional *cTLA* system (`CompSystemInstance`) is an instance of a process type (`CompSystemType`) containing process type instances (e.g. `pt1i1`, `pt2i1`, …). Each process type (`PT1`, `PT2`, …) may contain or extend further process types.

Because such a model structure is not possible with *Promela*'s process types (`proctype`), extended and contained process types must be resolved prior to building the *Promela* system. This is done during the expansion phase. As an example, consider the expansion of an action from the IP-ARP model (cf. **Listing 1**). The compositional form of the action is given by the coupling of actions from the contained process type instances `bnA` (of process type `IpArpNode`) and `med` (`Media`). The expanded form of the action contains no process type instances. Instead, init code and variables from the instances have been merged directly into the generated expanded or *flat* system type (`ExpSystemType`). This allows the actions to be flat as well, i.e. to directly consist of the merged action code of the previously coupled instances.

```
// Original Action as defined in the Compositional System
snd_A( pkt: PacketT ) ::= bnA.snd( pkt ) AND med.in( pkt );

// Action after Expansion (Flat System)
snd_A( pkt: PacketT ) ::=
  pValidIf(pkt.sci, NA_MII)                    // guards
  AND pkt = bnA_ifs[pkt.sci - 1].spa.pkt
  AND bnA_ifs[pkt.sci - 1].usd = TRUE
  ...
  AND bnA_ifs[pkt.sci - 1].spa.usd' = FALSE   // effects
  ...
```

Listing 1. Compositional and Expanded Form of an Action

Starting from the flat system, code optimizations can be applied. **Section 6** describes a few optimizations optionally done by *cTLA2PC* during this phase.

Depending on the chosen output, either the *cTLA* code generation or the *Promela* code generation phase follows. The key step in the *Promela* code generation phase is the handling of actions and their parameters. Because of the expansion phase only a single, simple process instance is left in the system. This instance, however, still contains multiple, parameterized actions. Thus, all actions are embedded into a *Promela* non-deterministic `do` selection loop. The translation of the actions themselves, which are structured into *guard and effect* statements, can be done quite easily. Quantified guards (*cTLA* keywords `FORALL`, `EXISTS`) and effects (`UPDATEALL`) are special cases which have to be handled through the introduction of local loop blocks and corresponding temporary variables (*Promela* keyword `hidden`) in the code.

Still, action parameters have to be handled. They are implicitly existentially quantified in *cTLA*, i.e. if parameter values exist that satisfy the action's guards, the action is executable with this parameter setting. Action parameters are handled through the introduction of shared global variables and input generator processes. At first, we tried using *Promela channels* instead of shared variables, but simple global

variables proved to be more efficient. The actions' parameters are replaced by these variables. *Input generator* processes are used to allow the global variables to reach all possible values. The processes use the *randomness non-deterministic if* approach described in [Ruy01]. Different actions may (re-)use the same global variables and input generator processes, thus reducing the number of additional variables and processes. The described approach works fine and was successfully used in [RPK04], but is relatively costly in terms of possible transitions and – to a lesser extent – state space. In **Section 6** we discuss a more efficient approach to the handling of parameterized actions.

Finally, the *Promela* code generation follows. Thanks to the previously generated intermediate code, the *Promela* code is derived in a straightforward way. This concludes the translation process. Additional translation options, which are useful for special cases, are recognized by *cTLA2PC*. For example, the `--simulation` switch includes a control flow generator and symbolic action names into the *Promela* code. This allows scripted testing of partial execution sequences and symbolic choice of actions in *SPIN*'s interactive simulation mode. Furthermore, the `--trace-points` switch helps in mapping *SPIN's* verification results back to the *cTLA* model. It inserts extra trace statements for *cTLA* actions and parameters into the *Promela* model.

6 Optimizations

Our current tool version, *cTLA2PC* 2, supports several switches for applying different optimizations to the *Promela* code. Some of the low-level optimizations are inspired from [Ruy01]. Ruys describes the *bitvector* optimization. *SPIN* internally stores each element of a bit array as a byte. This may lead to an eightfold increase in the size of the state vector. The bitvector optimization maps up to eight elements of a bit array into a single `byte` and replaces element accesses with appropriate macros. With cTLA2PCs `--optbitarrays` and `--opt-bool2byte` switches, we implement a *generalized bitvector* optimization. Arrays of records with multi bit fields – possibly of different size – are mapped into arrays of `byte`. Furthermore, `bool`s are mapped into `byte`s as well, because *SPIN* internally stores each `bool` as a `byte`. Using this generalized bitvector optimization we were able to significantly reduce the state vector for both the IP-ARP and the RIP model (cf. **Table 1**).

Table 1. Effects of Different Optimizations for both the IP-ARP and RIP Models

Model	Optimization	State Vector
IP-ARP	Standard	250 Bytes
	Paramodulation	210 Bytes
	Generalized Bitvector	168 Bytes
RIP	Standard	448 Bytes
	Paramodulation	424 Bytes
	Generalized Bitvector	344 Bytes

Higher level optimizations can lead to even better results. Our models have a special structure because of their *cTLA* origin. This of course leads to particular optimization possibilities. A rewarding area for optimizations is the transformation of actions. During this transformation several new processes and variables for handling action parameters are created (cf. **Section 5**).

The *paramodulation* optimization makes use of coupling between parameters in *cTLA* system actions. Typically, some action parameters serve as output parameters of constituting process actions. Thus, value determining equalities exist. Using these equalities, parameters occurrences in the action can be substituted and the parameters can be removed from the action's parameter list. As an example of a slightly more complicated case, reconsider action `snd_A( pkt: PacketT)` from **Listing 1**, where `PacketT` is a record and an equality `pkt = bnA_ifs[ pkt.sci-1 ].spa.pkt` exists. Substituting `pkt` using this equality does not work, because the right hand side depends on the field `sci` of `pkt`. However, after a *parameter refinement* of `snd_A`, i.e. splitting its parameter `pkt` into its fields *`scn, sci, sha, dha, dat`* and transforming all guards and effects containing `pkt` accordingly partial paramodulation becomes possible. Now, equalities without dependencies exist for all fields of `pkt` except `sci`. Accordingly, all parameters but `x_pkt_sci` can be substituted in `snd_A`, leading to the final version `snd_A(x_pkt_sci)` with just one simple parameter.

Paramodulation optimizes a model with respect to two aspects. First, the number of shared global variables is reduced, inducing a smaller state space. Second, corresponding input generator processes are saved as well. This leads to fewer possible transitions and accordingly smaller search depths for checking action sequences. In both the IP-ARP and the RIP model, the state vector is clearly smaller after applying the paramodulation optimization (cf. **Table 1**).

Even with paramodulation, however, the larger RIP based scenario could not be analyzed by *SPIN*. Input generator processes for setting parameter values substantially increase the complexity of the *Promela* model. The state vector is only enlarged by a small amount (about 4 bytes per process), but the number of possible transitions is expanded greatly. For example, each setting of a parameter value requires at least one step. If an action requires several parameters, usually a separate setting step is required for each parameter. Furthermore, because all input generators run freely as separate processes, two types of *useless sequences* are possible. First, sequences setting parameters not used by (and not defined for) an action may occur. Second, the same parameter may be set in several consecutive steps, each time overwriting the previous value. Only the last step of such a sequence before an action execution determines the parameter value. Because *SPIN* must consider all possible sequences for model checking, however, it has to follow the useless sequence types as well.

To prevent these useless sequence types, we evaluated making input generators and actions more intelligent. For that purpose, we enhanced *cTLA2PC* to add code to the beginning of an action that sets enable flags only for the input generators associated with the used parameters. Each input generator resets its enable flag after setting a parameter value. This approach prevents both useless sequence types mentioned above. Unfortunately, this approach proved to be counterproductive anyway. The additional flags and their management overhead usually add more complexity to the model than is saved through the prevention of the useless sequences.

Consequently, we developed a radically different approach for handling parameterized actions: *unrolling input generators*. Following this approach, no input generators are created by *cTLA2PC*. Instead, parameters are unrolled by copying the actions and replacing the parameters with fixed values. For example, an action with two parameters p1, p2 has to be copied $|p1|\cdot|p2|$ times, where $|p_i|$, i=1, 2 is the cardinality of the type associated with parameter p_i. In the copies, the parameters are successively replaced with fixed values for all possible values. Because all variable types in *cTLA* (and *Promela*) are finite, the number of fixed actions replacing a parameterized action is finite as well.

Of course, the unroll optimization may lead to very large *Promela* specifications, but this only increases translation time[1]. We evaluated the effects of the unroll optimization with the afore-mentioned RIP scenario. For benchmark purposes a simple assertion was added to the `rcv` action of host H2. This assertion was analyzed using *SPIN* in breadth-first search mode. As **Table 2** shows, *SPIN* performs remarkably better with the *Promela* model generated using the unroll approach than with standard input generators.

Table 2. Effects of the unroll optimization in the RIP model

Optimization	State Vector	Stored States	Transitions	Depth	Memory
Standard	332 Bytes	1.19E+06	2.3E+08	14	203 Bytes
Unroll	316 Bytes	1.99E+04	1.8E+06	11	11 Bytes

As input generator steps are no longer needed, the search depth required for finding a violating path is reduced as is the number of possible transitions at each level. Furthermore, the state vector is decreased as well. The unroll optimization was the critical last step for the successful automated analysis of the RIP model with *SPIN*.

Finally, efficiency should be kept in mind right from the design phase of a model. There, the framework helps again. Derived models inherit ideas from efficient protocol implementation (cf. **Section 3**), thus saving buffers and actions right from the start.

7 Eclipse Integration

We aim to ease the application of our modeling and analysis approach through appropriate tool support and integration. The *Eclipse* workbench is a well-known, widely adopted "universal tool platform" [Ecl05]. It defines only a core set of services. A modern plug-in architecture [Bol03] allows extending and customizing *Eclipse*'s functionality. Current web directories contain over 700 *Eclipse* plug-ins, even if many are of an experimental nature. We engineered a prototypical integration of the *SPIN* and *cTLA2PC* tools into *Eclipse*.

Our integration is comprised of 8 *Eclipse* plug-ins (cf. **Figure 5**) implemented by 70 Java classes, totaling about 12,000 lines of code.

[1] We experienced some problems with very large models during SPIN or gcc translation (`yacc` stack overflow errors with the SPIN Windows port and `gcc` hangs during verifier translation) but could work around them.

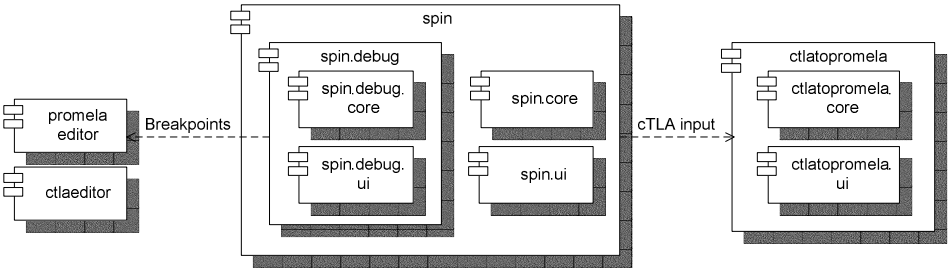


Fig. 5. Plug-in Architecture (UML Component Diagramm)

Except for the `promelaeditor` and `ctlaeditor` plug-ins, all plug-ins are separated into a `.ui` and `.core` component. User interface elements are implemented by the `.ui` component, the corresponding non-graphical functionality is implemented by the `.core` component. The underlying architectural pattern of the *Eclipse* framework is that different UI implementations can be used to present the same core functionality. Communication between UI and core components is handled via events.

Taken together, our plug-ins provide editing, translation, simulation, debugging and verification of specifications. Thanks to core services inherited from *Eclipse*, our integration covers further aspects, e.g. aggregation of files related to a specification into a project as well. For space reasons, we only describe the simulation and debugging of specifications in more detail.

To support simulation of *Promela* specifications from within *Eclipse SPIN*'s output is captured and transferred to *Eclipse*'s console window. Additionally, for interactive simulations, the output is parsed and an interactive selection dialog is displayed for each non-deterministic choice (cf. **Figure 6**). Choices marked by *SPIN* as "unexecuteable" are not displayed in the selection dialog. Furthermore, "debugging" of *Promela* specifications is supported as well. Breakpoints can be set in the *Promela* editor. If the corresponding line of the specification is hit, the simulation will be stopped. The user can then resume the specification simulation or step through it. Additionally, variables can be added to the watch window. That means that the current value of such a variable is always displayed by *Eclipse*.

The plug-in `spin.core` implements the functionality to run the *SPIN* tool in the background based on *Eclipse's Launching* architecture. For *SPIN* simulation a new `LaunchConfigurationsType` is defined. The `spin.ui` plug-in contains a dialog for setting additional *SPIN* options based on *Eclipse*'s `LaunchConfigurationDialog` and the selection dialog for interactive simulation. The `spin.debug.core` plug-in parses *SPIN* output and detects changes of watched variables, hit breakpoints etc. If breakpoints are defined, a `CodeModifier` is applied to the *Promela* file prior to starting the simulation. Its purpose is to add special marker `printf` statements at the appropriate lines. The plug-in captures *SPIN*'s output using a limiting buffer and scans it for the marker. If the marker is found, a breakpoint has been hit. The breakpoint's file and line number can be extracted from additional information after the marker. This implementation of breakpoints resembles *XSpin*.

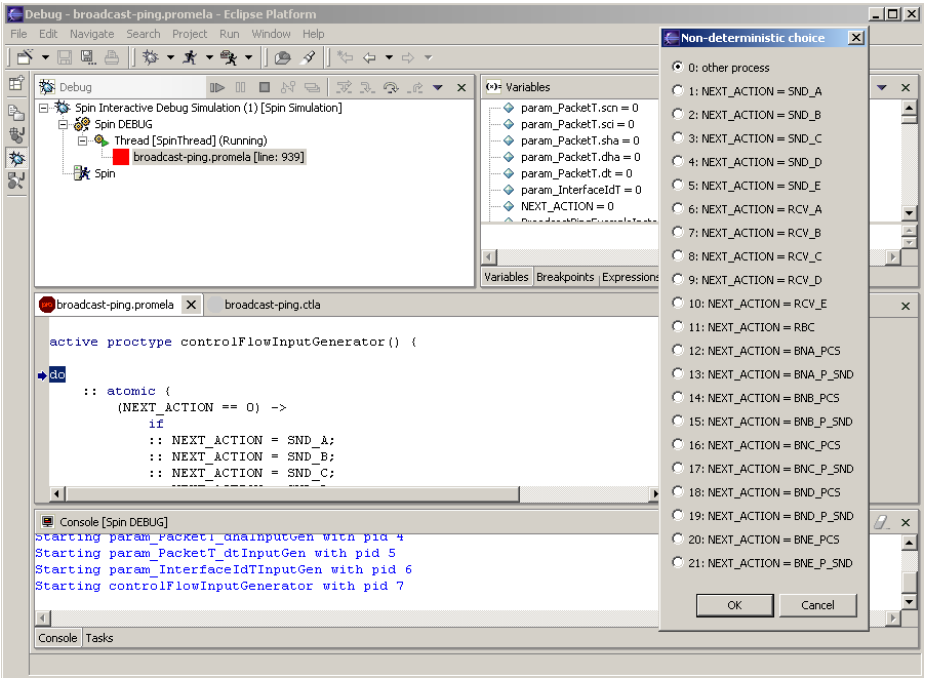


Fig. 6. Simulating a cTLA2PC generated Promela Specification in Eclipse

8 Concluding Remarks

The presented modeling framework and tool support facilitates the experimentation with small to medium size formal computer network models substantially and – as our experience showed – can be used not only for the precise description of known scenarios and attack processes but also for the automated detection of unknown attack variants. The development of models, however, is still a demanding task, since each model design decision about whether at all and how a certain detail of the real scenario is to be represented in the model, may yield either too strong an increase of the set of reachable states or the loss of relevant analysis results. Therefore, our current work continues to investigate approaches of efficient protocol implementation in order to achieve further enhancements of the modeling framework. Moreover, we study the integration of symbolic reasoning into the approach. Particularly symbolically proven state invariants shall help to justify model simplifications.

Another interesting idea - raised by the reviewers - is the application of our approach to areas not related to network attacks. The framework is specific for network modeling but otherwise our approach – high level *cTLA* and framework based modeling, optimized translation to *Promela*, model checking with *SPIN* – is generic. It would be interesting to apply it to other problems in the modeling, simulation, and analysis area.

References

[ABH97] R. Alur, R.K. Brayton, T.A. Henzinger, S. Qadeer, S.K. Rajamani: *Partial order reduction in symbolic state space exploration*. In Proc. of CAV 97: Computer-Aided Verification, LNCS 1254, pp. 340-351. Springer-Verlag, 1997.

[AP93] M. Abbott, L. Peterson: *Increasing network throughput by integrating protocol layers*. IEEE/ACM Transactions on Networking, pp. 600–610, 1, 1993.

[AR00] P. Ammann, R. Ritchey: *Using Model Checking to Analyze Network Vulnerabilities*. IEEE Symposium on Security and Privacy, May 2000.

[BAN89] M. Burrows, M. Abadi, R. Needham: *A Logic of Authentication*. In: Proceedings of the Royal Society, Volume 426, Number 1871, 1989.

[BHE01] Blackhat Europe Conference: *Routing and Tunneling Protocol Attacks*, URL: http://www.blackhat.com/html/bh-europe-01/bh-europe-01-speakers.html#FX, 2001.

[Bol03] A. Bolour: *Notes on the Eclipse Plug-in architecture*. URL: http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html

[BR00] M. Balser, W. Reif et al.: *Formal System Development with KIV*. In: T. Maibaum (ed.), Fundamental Approaches to Software Engineering. Springer LNCS 1783, 2000.

[Ecl05] Eclipse.org: *Main Page*. URL: http://www.eclipse.org, 2005.

[HK00] P. Herrmann, H. Krumm: *A Framework for Modeling Transfer Protocols*. Computer Networks, vol. 34, pp. 317-337, 2000.

[Hol03] G. J. Holzmann: *The SPIN Model Checker: Primer and Reference Manual*. Addison Wesley, 2003.

[KK03] K. Kawauchi, S. Kitazawa et al.: *A Vulnerability Assessment Tool Using First-Order Predicate Logic*. IPSJ SIGNotes Computer SECurity No.019 (2003)

[Lam94] L. Lamport: *The Temporal Logic of Actions*, ACM Transactions on Programming Languages and Systems, vol. 16(3), pp. 872-923, 1994.

[LBL99] G. Leduc, O. Bonaventure, L. Leonard et al: *Model-based Verification of a Security Protocol for Conditional Access to Services*. Formal Methods in System Design, Kluwer Academic Publishers, vol. 14(2), pp. 171-191, 1999.

[Mea96] C. Meadows: *The NRL Protocol Analyzer: An Overview*. Journal of Logic Programming, vol. 26(2), pp. 113-131, 1996.

[MS02] Paolo Maggi, Riccardo Sisto: *Using SPIN to Verify Security Properties of Cryptographic Protocols*. Proc. 9th Int. SPIN Workshop on Model Checking of Software, LNCS 2318, Springer-Verlag, pp. 187-204, 2002.

[NB02] S. Noel, B. O' Berry, R. Ritchey: *Representing TCP/IP connectivity for topological analysis of network security*. Computer Society, IEEE (ed.), Proc. of the 18th Annual Computer Security Applications Conference, pp. 25-31, 2002.

[RS02] C. Ramakrishnan, R. Sekar: *Model-Based Analysis of Configuration Vulnerabilities*. Journal of Computer Security, Vol. 10(1), pp. 189-209, 2002.

[RK03] G. Rothmaier, H. Krumm: *cTLA 2003 Description*. Technical Report, URL: http://ls4-www.cs.uni-dortmund.de/RVS/MA/hk/cTLA2003description.pdf, 2003.

[RPK04] G. Rothmaier, A. Pohl, H. Krumm: *Analyzing Network Management Effects with SPIN and cTLA*. Proc. of IFIP 18th WCC/SEC 2004, pp. 65-81, 2004.

[Rot04] G. Rothmaier: *cTLA Computer Network Specification Framework*. Online Document. URL: http://www4.cs.uni-dortmund.de/RVS/MA/hk/framework.html

[Ruy01] T. C. Ruys: *Towards Effective Model Checking*. PhD Thesis, University of Twente, 2001.

[Svo89] L. Svobodova: *Implementing OSI Systems*. IEEE Journal on Selected Areas in Communications 7, pp.1115–1130, 1989.

[Ver04] Verisign: *Internet Security Intelligence Briefing / November 2004 / Vol. 2 / Issue II.*

Model Checking Machine Code with the GNU Debugger

Eric Mercer and Michael Jones

Department of Computer Science,
Brigham Young University,
Provo, Utah, USA

Abstract. Embedded software verification is an important verification problem that requires the ability to reason about the timed semantics of concurrent behaviors at a low level of atomicity. Combining a cycle-accurate debugger with model checking algorithms provides an accurate model of software execution at the machine-code level while supporting concurrency and allowing abstractions to manage state explosion. We report on the design and implementation of such a model checker using the GNU debugger (gdb) with different processor backends. A significant feature of the resulting tool is that we can adjust the level of atomicity during the model checking run to reduce state explosion while focusing on behaviors that are likely to generate an error.

1 Introduction

Embedded software for small devices forms an important and unique verification problem. Embedded systems pervade many aspects of society, and their complexity is growing quickly with processing power. If processor design continues to follow Moore's law, then current test strategies will not be able to sufficiently validate safety and capital critical embedded systems.

Concurrency is a significant challenge to embedded software verification. Without fine-grain control of scheduling decisions in the operating system, it is not possible to explore the behaviors of concurrent interactions with typical debugging tools. A debugger is a familiar framework for software testing that faithfully reflects the behavior of the actual system because it is either running directly on the target hardware or on a high-fidelity back-end simulator. When running on native hardware, the debugger often uses hardware registers and traps, when necessary, to control program flow without significantly altering run-time behavior. Although a debugger provides several mechanisms to control program execution and alter program state, it does not provide mechanisms to adequately explore concurrent interactions. As such, a debugger is not sufficient to validate embedded software with concurrency due to threads, processes, or interrupts.

A model checker is well suited to the systematic exploration of concurrent behaviors. Several techniques for software model checking are actively being pursued. The most common approach applies conservative abstractions to the high-level programming language [9,1]. If no errors are found, then the program under

P. Godefroid (Ed.): SPIN 2005, LNCS 3639, pp. 251–265, 2005.

test is error-free (relative to the given specification, of course). Counterexamples may be either infeasible or feasible. Infeasible counterexamples are used to iteratively refine the abstraction and feasible counterexamples are returned to the user. This approach is particularly successful in verifying control intensive code in which conditional expressions do not depend on extensively manipulated data values. Another approach applies bounded model checking to C programs and can verify buffer overflows, pointer safety, user defined assertions, and language consistencies [4,5]. Other approaches translate the software under test into the formally defined input language of an existing model checker [15,17,2,6]. Language extensions are sometimes needed to facilitate the translation since the language semantics are not always supported by the existing framework [11].

With the exception of Bogor and Java PathFinder, each of the preceding approaches assume that the high-level language constructs are atomic operations. This assumption is adequate for properties that do not depend on machine instruction interleaving. In this work, we are interested in detecting errors that depend on a concurrency model that more accurately matches the behavior of programs running on a given target processor. This kind of concurrency model is particularly important for interrupt driven software. In most instruction set architectures, interrupts, due to concurrency or external inputs, can be taken between machine instructions, and many C instructions are often implemented with more than one machine instruction.

There are two approaches to software model checking that are directly pertinent to reasoning at a finer-grained level of concurrency. The first approach model checks the actual software implementation by instrumenting either a simulator or the virtual machine for the target architecture [20,13]. This approach retains a high-fidelity model of the target execution platform and low-level control of scheduling decisions. A second approach works directly with the machine-code of the program to run at-speed analysis on the native hardware. Valgrind instruments the actual machine code [14], and Verisoft runs the code in a scheduling environment as a process it manages [7]. Testing at speed boosts performance in state generation but can force processor, process, and OS specific specialization in the tool.

This paper proposes an approach to machine-code verification that uses a debugger to model check software compiled from various high-level languages. The approach is based on Java PathFinder, StEAM, and Verisoft but seeks to improve the processes by interfacing with a standard debugger rather than working through a virtual machine or interprocess communication mechanism. In other words, we use a debugger interface to control the program under test rather than a virtual machine for language and processor independence. And rather than manage the program under test in a scheduling processes, we manage the program under test in a debugger to have finer control of scheduling decisions and step levels. In essence, model checking with a cycle-accurate debugger provides a layer of abstraction that decouples the verification infrastructure from both the high-level source language and the processor simulation infrastructure.

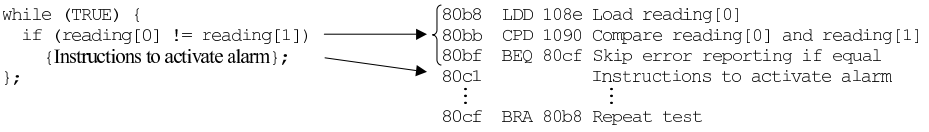


Fig. 1. The SSE example expressed in both C and machine code. The guard in the `if` statement is actually implemented with three machine-code instructions. Machine code generated by the GNU C compiler for the Motorola 68hc11 processor.

The central contribution of this paper is a better understanding of the challenges and opportunities of model checking machine code using a debugger. This understanding is based on our extension of the GNU debugger (gdb) which supports model checking for a variety of target processors at a variety of dynamically tunable atomicity levels. A debugger provides an accurate model of software execution at the machine-code level while allowing abstractions to manage state explosion. Furthermore, working at the machine-code level through a modular debugger decouples the model checker from a particular high-level language and target architecture. Note that the debugger does not solve the state explosion problem directly; rather, it provides mechanisms that can be used to mitigate state explosion by altering the atomic step level of the model checker. The combination of an accurate model with variable atomicity allows easy switching between less efficient model checking for error prone code sections and efficient model checking for less interesting code sections.

2 An Example

We illustrate the kinds of errors that can only be found at the machine-code level with a program, called SSE, that contains a simple serialization error. The error is manifested by a data inconsistency. Although the SSE program is somewhat naive, it illustrates the kinds of errors that can only be found when reasoning about concurrency at the machine-code level. In practice, more complex errors similar to the one in SSE arise when provably correct mutual exclusion techniques are either used or implemented incorrectly.

Figure 1 contains the C and machine-code versions of SSE. The machine code is a simplified version of the code generated by the GNU C compiler (gcc) for the Motorola 68hc11 processor. Simplifications are made strictly for readability in the figure. The analyzed code is the unmodified gcc output. The `while` loop in the C program contains an `if` statement that compares the readings of two sensors. If the readings are not equal, then an alarm is activated. The sensor readings are updated periodically by an interrupt handler (not shown) that copies readings from two input ports into the variables `reading[0]` and `reading[1]`.

In the assembly code for SSE, which is shown on the right side of Figure 1, the guard in the `if` statement is implemented with three instructions. The first instruction loads `reading[0]` (located at address 0x108e) into register D. The second instruction compares the contents of register D with `reading[1]` (located

at address 0x1090). The third instruction branches past the alarm activation code if the values are equal. If the interrupt to update `reading[0]` and `reading[1]` happens between the load and compare instructions, then the alarm may be incorrectly activated. The alarm may be incorrectly activated because one reading is stored in a register and the other in memory when the interrupt updates the contents of both in memory. Even if both readings are changed to the same value in memory, the now stale value in the register will be different. This particular interleaving is unreachable if the comparison in the guard in the `if` statement is modeled as an atomic comparison.

Of course, the serialization error in SSE can be eliminated by providing mutually exclusive access to the `reading` variables or by performing the check in the interrupt handler rather than in the busy-wait loop. If a lock is used to resolve the issue, then the lock can be implemented using any of a number of mutual exclusion algorithms that commonly appear as case studies in the model checking literature. The central verification issue addressed in this paper is not the correctness of mutual exclusion algorithms in general but the issue of whether or not a mutual exclusion algorithm is correctly implemented and used.

The SSE example can also be used to demonstrate the utility of including time in the processor execution model. The data inconsistency error in SSE can be eliminated by carefully scheduling the interrupts to occur only at safe locations between specific pairs of instructions. A fragment of machine code that preserves mutual exclusion using timing is shown in Figure 2. In the 68hc11, periodic interrupts are scheduled by writing a 16-bit value to a special timer "register" (two bytes stored at memory location 0x101c in this case) and setting a bit in a control register to enable the real-time interrupt. The interrupt is triggered when the free running counter is equal to the value written in the timer register. The free running counter is incremented by one in every clock cycle. The interrupt is serviced at the next instruction boundary after its corresponding interrupt flag is raised. The interrupt service routine clears the interrupt flag and schedules the next interrupt by writing a new value into the timer register. The new value is typically calculated by adding a fixed offset to the present value of the free running counter. Later, we compare the model checking results for the timed version of SSE with the behavior of the same program on the target hardware. In all cases, the predicted behavior precisely matches the actual behavior.

The execution of the guard and body of the `while` loop in SSE requires 22 clock cycles. Starting at the instruction on line 0x8053, which sets the next time-out value, the interrupt handler requires another 115 clock cycles before it returns control back to the interrupt point in the `while` loop. This timing relationship is shown in Figure 2. If the interrupt that updates `reading[0]` and `reading[1]` occurs with a period of $d = 22x + 115$ cycles (for values of x such that d is less than $2^{16} - 1$), then the update interrupt will alternate between the instructions at 0x80bf and 0x80cf. These are "safe" locations in which to update the `reading` variables. Another important aspect of this example is the amount of time that passes between the scheduling of the first interrupt and entry in the

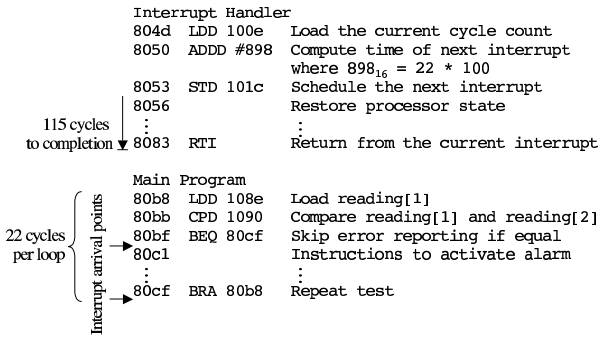


Fig. 2. Code that uses timing to avoid the serialization error in the SSE example

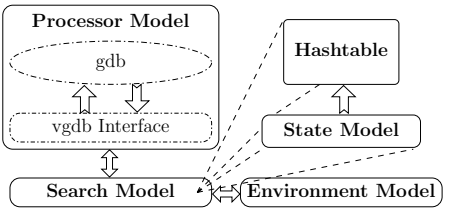
`while` loop. This delay fixes the location of the first interrupt in the `while` loop but is omitted from Figure 2 for clarity.

In practice, resolving mutual exclusion with time is advantageous because it does not require locking protocols (which can degrade important performance metrics such as response time). This approach is difficult to implement without automated verification support because it is conceptually difficult to correctly reason about timed concurrent behavior. A model checker that allows reasoning about timed concurrent behavior may extend design capacity by providing automated support for mutual exclusion that depends on timing. The goal of this work is to develop models and techniques to verify these types of systems and properties. Rather than create a processor model in an existing input language for a model checking tool, however, we are going to use the actual target processor hardware through a debugger.

3 State Enumeration by a Debugger

The state enumeration process is inspired by, and closely resembles, state generation in the Java PathFinder and StEAM tools in that they instrument virtual machines to perform model checking tasks [20,13]. In this case, the machine is real, not virtual, and the interface to the machine (or simulator) is a debugger. There are two principle tasks in state enumeration for explicit state model checking that the debugger needs to support: backtracking and resolving nondeterminism. In this section, we discuss the architecture of the resulting tools and how we deal with each of these tasks.

The general architecture for the model checker is shown in Figure 3(a). Roughly speaking, there are five major components to the architecture: the state, hash-table, processor, environment, and search models. We use the term "model" to indicate a generic representation of a particular component. The interface to each model is designed to allow flexibility in a manner similar to that of the Bogor framework [17].



(a)

```
Algorithm: bfs(s_o, PM, EM)
 1: init(Q, s_o)
 2: HT = {linearize(s_o)}
 3: while |Q| ≠ 0 do
 4:    s = dequeue(Q)
 5:    for all s' ∈ EM(s) do
 6:       s' = PM(s')
 7:       if linearize(s') ∉ HT then
 8:          if isViolation(s') then
 9:             reportTrace(s')
10:             return false
11:          HT = HT ∪ {linearize(s')}
12:          enqueue(Q, s')
13: return true
```

(b)

Fig. 3. The general architecture for model checking with a debugger and a search algorithm. (a) The general architecture showing the model checker interface to the debugger. (b) The breadth-first search algorithm using the processor model (PM), environment model (EM) and hash-table (HT).

3.1 State and Hash-Table Models

The state model represents the complete state of the processor and environment. It interfaces with the hash-table model using a *linearize* function that converts the state into an array of bytes. The state model is processor dependent; although, a generic configurable state model is provided that includes features commonly found in microprocessors such as general purpose registers, control registers, counters and memory. The interface exported by this model can be redefined to meet the verification requirements of a given program or target processor.

The hash-table model interfaces with the state model through the one-way linearize function which, as mentioned previously, converts the state into an array of bytes for storage. This split between the state model and the hash-table model decouples the representation of the state vector in state generation and storage for duplicate detection. Such a split simplifies the implementation of different hash-table architectures and storage disciplines. For example, super-trace (bit-state hashing) and hash-compaction can be implemented in the hash-table model without affecting the state model. The current implementation supports a collapse compression option [10].

Although the state and hash-table models simplify the implementation of symmetry and partial order reductions, the implementation of certain symmetry and partial order reductions are more difficult because these reductions require knowledge of the operating system data and how the data are structured in memory. For example, thread symmetry reduction would need to access operating system thread data to create the canonical state representation in the reduction. Note that this assumes two things: first, unfettered access through

the debugger to operating system data; and second, the ability to completely reload the state of the operating system in memory. An alternative approach bypasses the operating system and directly manages control structures for software artifacts [20,13,17]. This is done in our approach through the environment. Operating system calls to create threads and allocate memory can be tracked, or bypassed and rewritten, in the environment to make symmetry and partial order reductions easier to implement. Directly modeling, tracking, or abstracting the operating system are all possible in our approach using the debugger. The key to our approach is to be able to completely reload the machine state through the debugger interface which may only be possible when connected to a back-end simulator or remote target.

3.2 Processor Model

The processor model is the execution framework for the target architecture. In the current implementation, we use version 6.1 of the GNU debugger (gdb) for the execution framework. This version of the debugger is reconfigurable for cross-platform development using a collection of freely available back-end simulators. The actual input to the simulator, and hence the model checker, is a raw binary file in either an elf or a.out format. The binary file may or may not include debugging hooks to relate the machine-code to the high-level language from which it was compiled. The debugger can relate properly annotated machine code to a variety of high-level languages such as C, objective-C, C++, Pascal, Modula-2, Fortran and Ada.

The creation of a processor model is a pivotal point in the decision to create a new model checker rather than write processor models for use in another model checker, such as SPIN or Bogor. The central issue is the amount of work required to both create an accurate model of a processor architecture and implement the debugging features found in a debugger but not in any model checker.

The key feature found in a debugger, but not in existing model checkers, is the ability to change levels of atomicity during state generation. For example, state generation for a single code section might use C-instruction level atomicity for certain contexts and machine instruction level atomicity for others. Doing so in an existing model checker would require a significant rewrite. Currently, the actual steps the machine takes to update counters, process interrupts, etc. are invisible to the model checker. Moreover, we only record states at debugger break points, and these points can be defined in a variety of ways. The step level can be machine-code, high-level language, branch-point [3][1], or a mixed mode level, and all steps can be made conditional on run-time data values. Stepping at the branch-point level stops the debugger at points of nondeterminism that require an environment response. The mixed mode operates in any of the three levels and can be used to force the debugger to continue program execution until the program state satisfies a break point. This is useful in executing the program

[1] Branch-point refers to branches in the transition graph and not to branches in the program under test. In a deterministic environment, program branches do not produces branches in the transition graph.

across system calls or program states that are of no interest to the property being verified. Stepping through library calls allows one to accurately determine the effect of a library call on the property under test.

Modeling the processor in an existing framework is itself a challenge because some processor behaviors are neither simply described nor simply implemented. Presumably, leveraging the effort expended to create an accurate simulator rather than writing a new one from scratch frees one to focus on other issues. Some of the more difficult processor functions include interrupt priority resolution, interrupt register-stacking and control register updates. Other aspects of processor execution, such as instruction interpretation are straightforward if not monotonous.

The vgdb debugger component in Figure 3(a) defines the interface to gdb used by the model checker for explicit state enumeration. The interface takes a state model and loads it into the processor through the debugger. The debugger then turns control over to the target program which starts execution at the program counter in the loaded state. The debugger either steps at the machine-code level, the high-level language level, or until it runs to a breakpoint depending on the search model and user configuration. When the debugger stops, the model checker reads the resulting state from the debugger into the state model. Only modified parts of the state are updated in the state model. This saves time but still requires scanning the entire contents of memory in the debugger. This process can be further optimized with a map that identifies portions of memory that are either read-only, unaffected by the program under test, or simply out of bounds. This information is given to the model checker at runtime along with the program to verify and its properties.

The management of read-only and clear-on-write registers is a challenge in model checking with a debugger. The free running counter, used to track the passage of time in the 68hc11 processor, is an example of a read-only register. This register can only be set at boot time or when the processor is in test mode. The register that marks interrupt arrivals in the 68hc11 is an example of a clear-on-write register. It can only be cleared, not set, by a program.

When using a simulator, read-only and special control registers can be made writable by modifying the simulator. GNU gdb is well suited to this because it provides call-backs to implement all of the debugger functions in a simulator including a special interface to send commands directly to a simulator. For example, we modify the back-end simulator for the 68hc11 to include a command that puts the simulator in a mode that bypasses the write logic for special control registers such as those used for interrupt flags. When writing to these registers, rather than clearing the flags to acknowledge the interrupts, the simulator sets the flags.

Writing to read-only and special control registers can be simulated in hardware by carefully manipulating the state model. For example, as mentioned previously, the free running counter in the 68hc11 is read-only, but we need to control this register because it affects the firing of real-time interrupts. To address this issue, the state model is specialized to store the difference between

the current value of a timer register and the free running counter rather than the actual value of the timer register. When a state is loaded in and out of the debugger, the real-time interrupts are set to the current value of the real-time counter plus the difference stored in the state model. Similarly, when we read a state out of the debugger, we store the difference between the scheduled interrupt time and the current value of the real-time counter. Using this method, we are able to match real-time hardware interrupt behavior for the 68hc11. Another example relates to backtracking to states that have pending interrupts. Interrupt flags on the 68hc11 cannot be set. They can only be cleared. As mentioned earlier, we modify the simulator to let us set the flags, only this does not work when running on the native hardware. To accommodate this, we create a state that causes the actual interrupt to fire on the next machine instruction. This can be accomplished by carefully setting the real-time interrupts or toggling the external interrupt pin from the model checker through the serial or parallel port. Using this interface, the debugger can start execution from any arbitrary state in the program.

3.3 Environment Model

The environment model in Figure 3(a) closes the program under test by handling nondeterminism in a systematic way, checking invariants and configuring program dependent properties of the state. The environment model is implemented in C++ by the user and must be compiled into the model checker to create an executable that is specific to the program being verified.[2] More specifically, the environment provides a set of points to the model checker that represent either locations where an environment response is needed or locations where a property invariant needs to be checked. It is important to note that doing so does not require the source code for the program under test because these locations are instruction addresses. Although the environment model is implemented in C++, it can interface with code compiled from other high-level languages because the environment interfaces with the machine code emitted by the compiler rater than the original high-level source. Aside from the controls used by the debugger, the program runs unaltered in the model checker. Each environment response transforms the state model appropriately and returns the updated state to the model checker. The invariant checks are predicates defined over the variables in the state model which can include information specific to the environment.

Environment specific state information can be (and often must be) stored in the state model. For example, modeling thread scheduling in the environment requires environment specific state. When model checking a multithreaded program, it is possible to model check directly with the operating system, or it is also possible to abstract the operating system into the environment. If the operating system is abstracted, then the environment adds data to the state model to represent thread information. Instruction indices where we want to consider a possible scheduling operation are listed as points of nondeterminism in the

[2] Most of the model checker options are configured at run time on the command line in a manner very similar to that used in SPIN and Mur-ϕ.

environment model. The debugger stops at these indices, and the environment systematically generates states which explore the effects of different scheduling choices. Rather than providing a list of scheduling points, one could also use real-time interrupts in the target processor to implement round-robin scheduling. The interrupt handler can either implement a deterministic scheduling scheme or allow nondeterminism as before. Finally, the environment sets program specific state model properties. These properties might include read-only memory locations, track locations, match locations, and data abstractions.

3.4 Search Model

The final component for state enumeration is the search model that directs the traversal of the state space. Figure 3(b) shows pseudo-code for a breadth-first search model. The search model itself is an interface to the debugger which facilitates other search strategies. The breadth-first search example illustrates both the basic sequence of operations that might occur in a state enumeration strategy, and the interactions which might occur between various components of the system. In Figure 3(b), HT is the hash-table, EM is the environment model, and PM is the processor model. The breadth-first search does not use undo information to backtrack. Instead, the search maintains a queue, Q, of frontier states to be expanded. After a state is dequeued on line 4, it is sent to the environment for possible invariants checking and nondeterministic responses. If the environment does not have a response for the given state, then it is returned unaltered. Each environment response is sent to the processor model where it is loaded into the debugger, and the debugger begins execution. When the debugger stops, the new state is read, linearized, and sent to the hash-table for a membership check. If the new state is not a member of the hash table, then default properties are checked in line 8. These properties include stack overflow, read-only violations, and any properties general to most software programs.

The remainder of the pseudo-code in Figure 3(b) proceeds by updating the hash-table and adding the new state to the queue. The tool currently includes implementations of depth-first and guided-search in addition to breadth-first search. The tool implements the FSM distance heuristic and an extended FSM distance heuristic that is context sensitive for the guided-search [**?**,19].

4 Modeling Software

Model checking software requires extra care to properly handle functions, pointers, interrupts and external libraries. Model checking at the machine-code level simplifies the inclusion of functions, pointers, interrupts and libraries. This is primarily because the machine-code model must include all information needed to handle each of these unique properties of software. This is both good and bad because including such information further intensifies the state explosion problem. On the other hand, software model checking at the source-language level is somewhat more complicated because it must preserve the variables, function

calls and program flow abstractions provided to the programmer in a high-level language. These abstractions include the notions that variables range over all of the naturals (or reals) and that call stacks can be arbitrarily deep. Although reasoning about variables with infinite ranges is often simpler than reasoning about finitely ranged models, doing so can compromise the accuracy of the resulting analysis. The challenge is to retain accuracy while approaching the efficiency of infinite domain techniques.

Function calls are simplified because the calling context of every function is stored in the stack, which is part of the state model. There is a problem with recursive functions in which the depth of the stack is, theoretically, unbounded. In practice though, the stack is not unbounded and excessive recursive function calls eventually lead to stack overflow. In some verification settings, particularly for embedded software, determining a bound for the stack size and detecting such stack overflows is itself a significant problem [16]. The depth bound on function calls and recursion using a debugger is the same as it is in hardware.

Pointers can be handled in machine-code models because the state model is simply the contents of memory rather than a logical model of memory that requires alias analysis. The problem with pointers, for high-level language models, is that they can reference arbitrary memory locations, and this is difficult to model with a state vector that contains variables and their values (because the value of a pointer variable is an address of the value of interest rather than the value itself). A symptom of this problem is that when updating one variable's value, its not clear if any other pointer variables alias the same location and should also be updated. Working at the machine-code level eliminates this problem because there are no variables. There are only addresses and values in a large array. Essentially, we have the whole contents of memory in the state vector, so dereferencing a pointer and resolving aliasing issues is trivial during model checking. The new problem is that the state vector contains *all* of the memory locations, and this model may become large and difficult to update.

It should be mentioned that dynamic memory allocation, using a C-command like malloc, is also trivially modeled at the machine-code level. If the operating system is part of the state model, then the new memory is deterministically allocated according to the scheme implemented by the operating system. Otherwise, the environment model implements a possibly nondeterministic memory allocation scheme that mimics or approximates the actual memory allocation scheme. If the memory allocation scheme is deterministic, then pointers pointing to allocated memory can be compared across model checking runs. As with symmetry or partial order reductions, the implementation of symmetry reductions related to dynamic memory allocation, such as topological sorting in Bogor, can be implemented if care is taken to extract the appropriate information from either the environment or state model [18].

Interrupts allow program flow to transfer to an interrupt handler between any two instructions for which interrupts are enabled. Modeling interrupts in high-level language execution is somewhat awkward because an interrupt may appear during multiple machine instructions that implement a single high-level

instruction. In addition, modeling interrupts requires an accurate model of the interrupt timing and priority scheme of the target processor. This is impossible to do in the definition of a high-level language like C. At the machine-code level, an interrupt simply looks like another function call that is governed by the processor model rather than the program control flow. Using an accurate simulator, or even the target processor itself, provides a good model of interrupt behavior with no additional effort.

Library functions must be handled by either abstracting their behavior in the environment model or executing them directly as part of the state model. If library functions are included in the state model, then the environment model can be written to note the library calls and then step over them so that the effects are computed but the states reached in them are not stored or checked. Some library calls in some situations may be of interest. In these cases, conditional breakpoints can be set to step into the library calls and include their behavior in the verification run when needed.

5 Results

We have implemented the model checking architecture described in Section 3 in a tool called Estes. Estes is implemented in C++ as an extension of gdb. The implementation required approximately 8 man months of effort. The development effort thus far is focused on correctness and has yet to consider performance in terms of raw state generation per unit time. We believe state generation can be improved through profiling and appropriate code optimizations to the point where the principle performance penalty is that imposed by the debugger itself.

The first results use the SSE program to validate the accuracy of the model checker relative to the actual hardware. The program in Figure 1 does not specify when interrupts may or may not fire so we conservatively assume that they can fire between any pair of C or machine code instructions depending on the verification step level. As expected, if the verification is performed at the C language level, then the alarm is always correctly activated because we do not allow interrupts inside the guard of the `if` statement. If the verification is performed at the machine-code level, then the alarm is incorrectly activated.

The SSE program in Figure 2 uses real-time interrupts to remove the incorrect activation of the alarm. The verification is configured to step-over functions that update the LCD register. Doing so results in a significant savings in time and space because the code to write to the LCD implements delays using while-loops that require 2^{16} iterations to complete. We also instrument the SSE code to count the number and location of the interrupts, as well as the interrupt sequence. The results from the debugger tool and the actual hardware match exactly on all points.

The next set of results simply demonstrate some applications of Estes to C programs and an Ada program. The results are from implementations of a serial IO program, the Hyman [12] and Peterson mutual exclusion algorithms and a classic dining philosophers. The serial IO program reads from the serial IO port

Table 1. Results for the Motorola 68hc11 processor

	High-level language			Mixed		Assembly language		
Program (language)	lines	time	states	time	states	lines	time	states
Serial IO (Ada)	25	0.03s	15	-	-	116	3.7s	17441
Hyman (C)	80	0.5s	387	2.2s	2005	255	5.9s	6948
Peterson (C)	80	1.2s	655	48s	19423	4443	45s	31772
Dining Philosopher 2 Interrupts (C)	295	0.8s	1520	-	-	595	2.8s	7722
Dining Philosopher 3 Threads (C)	150	-	-	2.93s	10667	468	108s	382359

and writes the input back to the serial IO port. The environment model for the Serial IO program, which is written in C++ and references the machine code created from the Ada source, allows the input values to range over 256 values. The Hyman and dining philosopher algorithms both have errors that can be detected at the high-language level while the Serial IO and Peterson programs do not contain errors. There are two versions of dining philosophers using either interrupts or threads.

The results from a Pentium III 1.8 GHz processor with 1 GB of RAM are shown in Table 1. The machine code is again compiled for a 68hc11 chip with the debugger connected to a back-end simulator. We show results for high-level, assembly code, and mixed levels. The high-level results allow interrupts only at Ada or C instruction boundaries. The mixed level steps over unimportant code, such as the LCD output in the Petersons example, but allows interrupts at the machine-code instruction boundaries. For the results at the high-level and assembly language, the *lines* columns are the total number of lines of code from either the high-level language file or the file created from an object dump of the binary executable as indicated by the top-level column division. The *time* columns contain the wall-clock time measured using the Unix time program which includes actual start up and shutdown overhead for the model checker and debugger. The *states* columns give the total number of states found using a depth-first search when the search either finds an error or exhausts the state space.

The results for the Hyman and Peterson programs in Table 1 are generated with the ideal switchings between the assembly and C step-levels. The switching points are statically defined by the user but can be easily configured from run to run without recompilation. It is not always easy or feasible to construct the switching protocol by hand. Future work explores heuristic methods for switching between step-levels during model checking depending on the property under test.

6 Conclusion

Model checking at the machine-code level using a debugger results in an accurate, timed model of the software under test. As expected, allowing concurrency at the machine-code level allows the detection of errors that are missed at the C-

instruction level and further compounds the state explosion problem. Switching between levels of atomicity on-the-fly based on conditions evaluated at run-time allows one to focus verification effort on (and to contain state explosion within) specific regions of specific execution paths in the software under test. Modeling software at the machine-code level simplifies handling some problems unique to software model checking such as pointers, function calls, interrupts and library functions because all of the information needed to resolve such issues is included in the state model. This also exacerbates the state explosion problem.

Future work on model checking machine code with a debugger focuses on methods for containing the state explosion problem. A variety of state analysis techniques can be applied to machine code to simplify model checking. Static analysis of machine code is difficult because variables do not always have well defined types and scopes. Since we have a fully executable state vector during model checking, we can pause a model checking run to redo the parts of the initial static analysis using the concrete information in the state vector. The updated static information can be used for guided search heuristics and dead variable analysis to reduce the size of the state vector. Finally, the incorporation of symbolic techniques using BDD or SAT algorithms using a variant of the track and match methodology may provide a further increase in model checking capacity.

References

1. T. Ball and S. Rajamani. The SLAM toolkit. In G. Berry, H. Comon, and A. Finkel, editors, *13th Annual Conference on Computer Aided Verification (CAV 2001)*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264, Paris, France, July 2001. Springer-Verlag.
2. T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In Klaus Havelund, John Penix, and Willem Visser, editors, *7th International SPIN Workshop*, volume 1885 of *Lecture Notes in Computer Science*, pages 113–130. Springer, August 2000.
3. G. Behrmann, K.G. Larsen, and R. Pelánek. To store or not to store. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Proc. Computer Aided Verification (CAV'03)*, volume 2725 of *Lecture Notes in Computer Science*, pages 433–445, Boulder, CO, USA, July 2003. Springer.
4. E. Clarke and D. Kroening. Hardware verification using ANSI-C programs as a reference. In *Proceedings of ASP-DAC 2003*, pages 308–311, Yokohama City, Japan, January 2003. IEEE Computer Society Press.
5. E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176, Barcelona, Spain, April 2004. Springer.
6. J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, R. Zheng, and H. Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.
7. P. Godefroid. Software model checking: The VeriSoft approach. Technical report, Bell Laboratories, Lucent Technologies, 2003.

8. S. Graf and L. Mounier, editors. *Model Checking Software: 11th International SPIN Workshop*, volume 2989 of *Lecture Notes in Computer Science*, Barcelona, Spain, April 2004. Springer.
9. T. A. Henzinger, R. Jhala, R. Majumdar, , and G. Sutre. Software verification with Blast. In T. Ball and S.K. Rajamani, editors, *Proceedings of the Tenth International Workshop on Model Checking of Software (SPIN)*, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239, Portland, OR, May 2003.
10. G. J. Holzmann. State compression in Spin. In *Proceedings of the Third Spin Workshop*, Twente University, The Netherlands, April 1997.
11. G. J. Holzmann and R. Joshi. Model-driven software verification. In Graf and Mounier [8], pages 76–91.
12. H. Hyman. Comments on a problem in concurrent programming control. *Communications of the ACM*, 9(1):45, 1966.
13. T. Mehler and S. Edelkamp. Directed error detection in C++ with the assembley-level model checker StEAM. In Graf and Mounier [8], pages 39–56.
14. N. Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, Computer Laboratory, University of Cambridge, United Kingdom, September 2004.
15. J. Penix, W. Visser, C. Pasaranu, E. Engstrom, A. Larson, and N. Weininger. Verifying time partitioning in the DEOS scheduling kernel. In *22nd International Conference on Software Engineering (ICSE00)*, pages 488–497, Limerick, Ireland, June 2000. ACM.
16. J. Regehr, A. Reid, and K. Webb. Eliminating stack overflow by abstract interpretation. In R. Alur and I. Lee, editors, *Proceedings of the Third International Conference on Embedded Software (EMSOFT)*, volume 2855 of *Lecture Notes in Computer Science*, pages 306–322, Philadelphia, PA, USA, October 2003. Springer.
17. Robby, M. B. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular model checking framework. *ACM SIGSOFT Software Engineering Notes*, 28(5):267–276, September 2003.
18. Robby, M. B. Dwyer, J. Hatcliff, and R. Iosif. Space-reduction strategies for model checking dynamic software. *Electronic Notes in Theorical Computer Science*, 89(3), 2003.
19. N. Rungta and E. G. Mercer. A context-sensitive structural heuristic for guided search model checking. `http://vv.cs.byu.edu/publications/papers/guided-search.pdf`, 2005.
20. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2), April 2003.

ETCH: An Enhanced Type Checking Tool for Promela

Alastair F. Donaldson* and Simon J. Gay

Department of Computing Science,
University of Glasgow,
Glasgow, Scotland
{ally, simon}@dcs.gla.ac.uk

Abstract. We present ETCH, an enhanced type checking tool for the Promela language. This tool uses standard type checking in conjunction with constraint-based type inference to detect type errors in Promela models which cannot currently be detected by SPIN before verification or simulation. ETCH allows for more rapid development of Promela code, and increased confidence in verification models used with SPIN. Since the utility of model checking depends heavily on the correctness of the model being verified, our tool is a significant contribution.

1 Introduction

Model checkers and type checkers have both been classed as *light weight* formal methods [9]. Type checkers for high level languages have been widely used in everyday software development for many years, and model checkers are increasingly being used in the development of reliable hardware and software. Verification models for use with a model checker are usually written in a high level language, and if this language includes type information, type checking can be used to aid the development of sensible models. The Promela language [6] includes a rich set of data types, including numeric types, enumerated message types, and types for communication channels. Before simulation or verification of a Promela model, the SPIN model checker performs some type checking to detect errors in the model. However, the type checking performed by SPIN is limited. Certain kinds of type errors which are not currently detected by SPIN could be detected in a straightforward manner using the type information included in a Promela model. More subtle errors involving dynamic channel passing (an attractive feature of the language) cannot be detected directly from this type information, since channel types in Promela are not fully specified. To be detected statically, such errors require additional type information to be inferred from the model.

We present ETCH (Enhanced Type CHecker), a type checking tool for Promela. This tool uses standard type checking in conjunction with constraint-based type inference to detect type errors in Promela models which cannot currently be

* Supported by the Carnegie Trust for the Universities of Scotland.

P. Godefroid (Ed.): SPIN 2005, LNCS 3639, pp. 266–271, 2005.

detected by SPIN. ETCH allows for more rapid development of reliable Promela code, and increased confidence in verification models used with SPIN. Our approach requires no modification to the syntax of Promela, and no extra type declarations are necessary—type checking is performed by type inference based on the existing type information in a Promela specification. Programmers in any language know that type errors are a frequent kind of mistake, and that compile-time type checking is very useful. Promela is no exception. For example, in an informal survey of Promela code produced for a student assignment, ETCH was used to detect numerous type errors which were not detected by SPIN. Since the utility of model checking depends heavily on the correctness of the model being verified, our tool is a significant contribution.

In Section 2 we give some examples of type errors in Promela code which are not detected by SPIN until verification time. In Section 3 we outline the design of ETCH. We discuss two interesting features of the tool—constraint-based type inference, and recursive channel types—in Sections 4 and 5 respectively. Conclusions and plans for future work are given in Section 6. ETCH can be downloaded from our website [3].

2 Example

To illustrate the kind of type errors which currently are not detected by SPIN before verification time, we consider a generic client-server model adapted from [6, Chapter 15]. The Promela code for this model is given below, annotated with asterisks which are for discussion purposes, and should otherwise be ignored.

```
mtype = {request,deny,hold,grant,return}
chan server = [0] of {mtype,chan}
chan null = [0] of {mtype,chan}

proctype Agent(chan listen, talk)
{ do
   :: talk!hold(listen)           (**)
   :: talk!deny(listen) -> break
   :: talk!grant(listen) ->       (***)
wait: listen?return(null); break
  od;
  server!return(listen)   }

active[2] proctype Client()
{ chan me = [0] of {mtype,chan};
  chan agent;
end: do
      :: timeout ->
         server!request(me);
         do
          :: me?hold(agent)
          :: me?deny(agent) -> break
          :: me?grant(agent) ->
             agent!return(null); break
         od
  od   }

active proctype Server()
{
  chan agents[2] = [0] of {mtype,chan};
  chan pool = [2] of {chan};
  chan client, agent;
  byte i;
  do
   :: i < 2 -> pool!agents[i]; i++
   :: else -> break
  od;
end:
  do
   :: server?request(client) ->
      if
       :: empty(pool) -> client!deny(null)
       :: nempty(pool) -> pool?agent;
          run Agent(agent,client)          (*)
      fi
   :: server?return(agent) -> pool!agent
  od
}
```

We now suggest three changes to the above model which introduce type errors.

Error 1. *The statement* `run Agent(agent,client)` *at* `(*)` *is replaced with the statement* `run Agent(agent)`. This is clearly a type error since the *Agent*

proctype requires two parameters and only one has been supplied. The SPIN syntax checker does not detect this error. If this statement is executed during simulation then the message `Error: missing actual parameters: 'Agent'` is given, and simulation halts. During verification, an *Agent* process will be instantiated on execution of this statement, but the *talk* parameter of the *Agent* will be an uninitialised channel, resulting in an error from SPIN when this channel is used. ETCH detects this error without needing to use type inference.

Error 2. *The statement* `talk!hold(listen)` *at* `(**)` *is replaced with the statement* `talk!listen(hold)`. From the model, we can see that *Agent* processes are instantiated only by the *Server* process. The *talk* parameter of an *Agent* corresponds to the *client* variable of the *Server* process, and this is in turn received on the channel *server*, from a *Client* process. The channel which the *Client* process sends is *me*, which accepts messages of the form $\{mtype, chan\}$. Thus the *talk* parameter of an *Agent* accepts messages of the form $\{mtype, chan\}$. Our modification introduces a type error since an attempt is made to send a message of the form $\{chan, mtype\}$ on the channel *talk*. This error is not picked up by the SPIN syntax checker. During simulation, SPIN reports a type-clash if this statement is executed. An invalid end state is reported during verification of the model with this error, and the corresponding counterexample contains a warning of a type-clash. In Section 4, we describe how ETCH detects this error using constraint-based type inference.

Error 3. *The statement* `talk!grant(listen)` *at* `(***)` *is replaced with the statement* `talk!grant`. By the same argument presented for Error 2, this modification causes an error since only a single field has been sent on the channel *talk*. This error is not detected by the syntax checker. Since the argument *grant* has type *mtype*—the correct type for the first message field—SPIN does not flag up an error when this statement is executed, even though a field is missing from the message (during simulation a warning is given). The message is received by a *client* process via the statement `me?grant(agent)`. However, the received channel *agent* is uninitialised since no channel was actually sent by the *Agent* process. The *Client* process then attempts to execute the statement `agent!return`, which causes an error since *agent* is not initialised. This error is less easy to isolate than Errors 1 and 2 since it has effect at a later stage in system execution. In Section 4 we describe how ETCH detects this error using constraint-based type inference.

ETCH detects each of these errors statically, before simulation or verification of the model, making it easier to eliminate them.

3 Overview of ETCH

We have implemented ETCH in Java, using the compiler generation framework SableCC [5] to generate a parser for Promela based on the grammar provided in [6]. The type system used by ETCH is based on type systems for the π-calculus

[10, Chapter 6]. The core grammar which ETCH uses to represent types is as follows:

$$
\begin{array}{lll}
T ::= \mathsf{numeric\ type} \mid \mathsf{pid} \mid \mathsf{mtype} \mid \mathsf{bool} & & \textit{basic types} \\
\quad \mid \mathsf{chan}\, C & & \textit{channels} \\
\quad \mid \mu X.T & & \textit{recursive types} \\
C ::= X & & \textit{type variables} \\
\quad \mid \{T_1, \ldots, T_n\} & & \textit{tuples}
\end{array}
$$

Array and record types are also supported, with the same restrictions as are imposed by SPIN. In themselves they do not generate any complications for type checking. Type checking of Promela code is performed by ETCH using the type information included with variable declarations. Each time a channel declaration is encountered, ETCH chooses a fresh type variable to represent the type of messages which may be sent on the channel. Constraints on the form of type variables are generated based on applied occurrences of channel identifiers. A standard constraint-based type inference algorithm [1, Chapter 6] is used to solve these constraints in order to determine values for the type variables. Recursive channel types are introduced in order to solve constraints of the form $X = \mathit{chan}\, C$ where X occurs in C. We discuss constraint-based type inference and recursive channel types in Sections 4 and 5 respectively.

Promela has a variety of numeric types representing different numeric ranges, including a type of unsigned integers which is parameterised by the word length. ETCH implements the natural subtyping relations, e.g. $\mathit{byte} <: \mathit{unsigned}(12) <: \mathit{short} <: \mathit{int}$. For programming convenience, $\mathit{bit} <: \mathit{bool}$, although *bool* is not related to other numeric types. By default, ETCH (like SPIN) treats the types *pid* and *byte* as equivalent, but there is an option to regard them as different types. This is useful in e.g. work by the first author on symmetry detection [4].

Since ETCH is not part of SPIN, any errors reported by ETCH are really just warnings. Therefore we have taken a strict approach in deciding what constitutes a type error. SPIN treats enumerated *mtype* variables and values as if they were numeric, and thus allows *mtype* variables and values to be used in any context in which numeric variables and values can be used. For example, if a, b and c are *mtype* variables, SPIN allows a statement such as $a = b + c$. This use of message types in an arithmetic context is usually bad practice—arithmetic on *mtype* values is also disallowed in [7]—and so ETCH will report a type error in such cases. Running ETCH on the client-server model, modified to include Error 1, results in the following error message:

```
Line 47: Error - the proctype "Agent" expects 2 arguments but
1 has been supplied
```

The message corresponding to Error 3 is:

```
Line 9: Error - arguments of different lengths have been used
for the same channel.  Unable to unify "{mtype,chan X7}"
and "{mtype}"
```

Here `X7` is the name of a type variable used by the type inference algorithm (see Section 4). Error 2 generates a similar warning.

4 Constraint-Based Type Inference

Channel types are only partially specified in Promela. For example, the declaration `chan server = [0] of {mtype,chan}`, in the client-server model of Section 2, specifies that the second field of a message to be sent on *server* should be a channel, but does not specify the type of this channel. ETCH represents channel types fully, using type-variables for types which are not known (see the grammar for types presented in Section 3). To illustrate the approach of constraint-based type inference used by ETCH, consider the client-server example, modified to include Error 2. The `listen` and `talk` parameters of the *Agent* proctype are assigned types $chan\,X$ and $chan\,Y$ respectively, since the types of messages which they accept are not specified. The (modified) statement `talk!listen(hold)` causes the constraint $chan\,Y = chan\{chan\,X, mtype\}$ to be stored, while the statement `talk!deny(listen)` causes the constraint $chan\,Y = chan\{mtype, chan\,X\}$ to be stored. Attempting to unify these constraints results in the constraint $chan\,X = mtype$, which cannot be unified, thus a type error is generated.

Now consider the client-server example modified to include Error 3. Again, `listen` and `talk` are assigned types $chan\,X$ and $chan\,Y$ respectively. The statements `talk!hold(listen)` and `talk!deny(listen)` both result in the constraint $chan\,Y = chan\{mtype, chan\,X\}$. However, the (modified) statement `talk!grant` results in the constraint $chan\,Y = chan\{mtype\}$. Unification of these constraints fails since the tuples $\{mtype, chan\,X\}$ and $\{mtype\}$ cannot be unified, being of different lengths.

For a more general description of constraint-based type inference, see [9, Chapter 22]. Our implementation is based on an algorithm described by Aho et al. [1, Chapter 6].

5 Recursive Channel Types

Consider the following Promela code: `chan A = [1] of {chan,bit}; A!A,0`. The channel A accepts messages with two fields. The second field should be of type *bit*, and the first must be a channel of *the same type as A*. This because the channel A is sent on itself by the statement `A!A,0`. Thus the type of channel A is recursive, and using standard notation, we can express the type of A as $\mu X.chan\{X, bit\}$. The above Promela code fragment is legitimate, and this kind of channel usage has been employed in realistic Promela models, e.g. a model of a telephone system [2]. We have incorporated recursive types into ETCH. A recursive type expression has infinitely many equivalent syntactic representations, depending on where the recursive μ construct appears in the expression, and on how far the type expression has been *unfolded*. A recursive type expression resulting from the unification of a set of constraints may look very complex, even

though it is equivalent to a much shorter expression. When presenting types to the user in type errors, it is desirable to convert recursive types into their simplest forms. ETCH incorporates a minimisation algorithm for recursive types. This algorithm is based on an algorithm for minimising a deterministic finite automaton [8, Chapter 2], and involves finding the largest bisimulation on a type expression. For example, the types

$$\mu X.chan\{chan\{X, bit\}, bit\} \quad \text{and} \quad chan\,\mu X.\{chan\,X, bit\}$$

are the same, and are equivalent to the type $\mu X.chan\{X, bit\}$, which is the minimal form to which ETCH converts both type expressions.

6 Conclusions and Future Work

We have presented ETCH, an enhanced type checker for Promela. ETCH extends the capabilities of SPIN by detecting type errors in a model before simulation or verification. Eliminating errors using ETCH results in more reliable verification models for use with SPIN, and thus increased confidence in SPIN verifications. The standard Promela language is handled in full by ETCH. Inline macros, not part of the language grammar given in [6], cannot be handled at present. Extending ETCH to handle inline macros should be straightforward. We also intend to improve the quality of error messages arising due to errors detected by the type inference algorithm.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers—Principles, Techniques and Tools*. Addison-Wesley, 1986.
2. M. Calder and A. Miller. Using SPIN for Feature Interaction Analysis - a Case Study. In *Proceedings of the 8th International SPIN Workshop on Model Checking Software*, LNCS 2057, pages 143–162. Springer, 2001.
3. A. F. Donaldson and S. J. Gay. ETCH Website: `http://www.dcs.gla.ac.uk/people/personal/ally/etch/`.
4. A. F. Donaldson and A. Miller. Automatic Symmetry Detection for Model Checking Using Computational Group Theory. In *Proceedings of the 13th International Symposium on Formal Methods*, LNCS 3582, pages 418–496. Springer, 2005.
5. E. Gagnon and L. J. Hendren. SableCC, an object-oriented compiler framework. In *26th Technology of Object-Oriented Languages and Systems*, pages 140–154. IEEE Computer Society Press, 1998.
6. G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.
7. S. Leue, R. Mayr, and W. Wei. A Scalable Incomplete Test for Message Buffer Overflow in Promela Models. In *Proceedings of the 11th International SPIN Workshop on Model Checking Software*, LNCS 2989, pages 216–233. Springer, 2004.
8. P. Linz. *An Introduction to Formal Languages and Automata*. Jones and Bartlett, 1986.
9. B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
10. D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

Enhanced Probabilistic Verification with 3Spin and 3Murphi

Peter C. Dillinger and Panagiotis Manolios

College of Computing, Georgia Institute of Technology,
801 Atlantic Drive, Atlanta, GA 30332-0280
{peterd, manolios}@cc.gatech.edu

Abstract. 3Spin and 3Murphi are modified versions of the Spin model checker and the Murφ verifier. Our modifications enhance the probabilistic algorithms and data structures for storing visited states, making them more effective and more usable for verifying huge transition systems. The tools also support a verification methodology designed to minimize time to finding errors, or to reaching desired certainty of error-freedom. This methodology calls for bitstate hashing, hash compaction, and integrated analyses of both to provide feedback and advice to the user. 3Spin and 3Murphi are the only tools to offer this support, and do so with the most powerful and flexible currently-available implementations of the underlying algorithms and data structures.

1 Introduction

Explicit-state model checking is a popular and effective verification technique employed by numerous tools, including Murφ, TLC, Java PathFinder, and Spin. To ameliorate the memory demands of state explosion, most of these tools include algorithms that have a small probability of overlooking errors. One such probabilistic algorithm is *bitstate hashing*, developed in Spin [6]. The other major probabilistic technique is *hash compaction*, which was mostly developed in Murφ [8,9] (building on previous work [10]). We have modified these two tools into releases called 3Spin [3] and 3Murphi [2] that each incorporate a set of features designed to maximize the effectiveness and efficiency of probabilistic verification (see Table 1). This novel set of features is utilized by our probabilistic verification methodology, which was introduced in [4] but

Table 1. This table shows the capabilities of four probabilistic explicit-state verification tools, Murφ 3.1, Spin 4.2.2, 3Murphi 3.2, and 3Spin 3.2

Tool	Bitstate hashing	Hash compaction	Memory sizes	Hashing	Feedback
Murφ	no	yes, with wasted bit	any	univ.+diff.	H.C. omission analysis (probability only)
Spin	enhanced ($\geq$ 4.2.0)	yes, but inflexible	powers of 2 only	Jenkins	B.H. recommendations ($\geq$ 4.2.0)
3Spin, 3Murphi	enhanced	yes	any	Jenkins; univ.+diff.	recommendations (both algs); omission analysis (both algs)

P. Godefroid (Ed.): SPIN 2005, LNCS 3639, pp. 272 – 276, 2005.

we review here in Section 2. Sections 3 and 4 then discuss how the features of 3Spin or 3Murphi and our methodology enable the user to more efficiently reach her verification goal while preserving all existing functionality of Spin and Murphi (respectively).

2 Methodology

In previous work [4] we describe a methodology for utilizing both bitstate hashing and hash compaction that attempts to minimize the time to finding errors (if present) or to reaching whatever certainty the user considers adequate to concluding that the model satisfies the desired properties.

Bitstate hashing and hash compaction are probabilistic data structures used to represent sets. They support the standard *add* and *query* operations, but a query on an element that is not in the set may return *true*, yielding a false positive. Their probabilistic nature allows for memory-efficient representations of large sets, a crucial requirement for model checkers which have to keep track of very large sets of visited states. Bitstate hashing identifies states with a chosen number of addresses of a bit-vector; when a state is visited the corresponding bits are set. Hash compaction stores hashed states in a table with a fixed number of cells. Figure 1 emphasizes the key difference between the data structures, which we expand upon in our methodology description below.

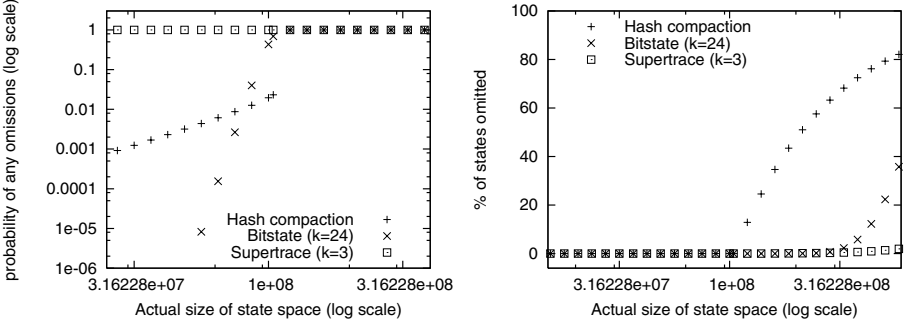


Fig. 1. These graphs show the accuracy of three probabilistic verification techniques/configurations for various state space sizes. In both graphs, lower is better. The data points for "Hash compaction" and "Bitstate (k=24)" are obtained with data structures optimized for a state space size of 10^8, using 400MB of memory. The graphs show the accuracy of the data structures as the size of the state space varies. In both graphs, lower is better according to the respective criterion. The left graph shows the probability that any omissions occur, while the right graphs shows the expected percentage of states omitted. "Bitstate (k=24)" and "Supertrace (k=3)" are instances of bitstate hashing setting 24 and 3 bits per state respectively. "Hash compaction" shows the expected results of using 32-bit hash compaction, which in this case has a maximum visitable size of about 105 million states. The results are obtained mostly analytically, as in [4].

When the state space size is completely unknown, as when first attempting to verify a model, use *supertrace*, which is bitstate hashing configured to set a small number

of bits per state, such as Holzmann's recommendation of three. In such cases, supertrace is the best choice because of its low percentage of omitted states over a wide range of state space sizes (see right graph in Figure 1). Supertrace tends to find errors quickly if they exist, but is not the most productive technique for demonstrating error-freedom [5].

When we know the size of the state space rather accurately, as when iteratively building confidence of error-freedom in a model, use *hash compaction*, because of its superior accuracy when tuned for a known state space size. The left graph of Figure 1 shows that when the actual state space size is 80–100% of the maximum size, hash compaction is the most accurate. The graphs also show that if the table ends up far from full (left 1/3rd of left graph) or if it overflows (right half of right graph), hash compaction is not the best choice.

When we have a rough estimate of the state space size, as when verifying a modified version of a previously-verified model, use *bitstate hashing* configured to set a number of bits per state optimized for that estimate (shown in [4]). When that number is significantly larger than supertrace's 3 bits, this approach is likely to be much more accurate. Furthermore, it can tolerate much more deviation from the estimate than hash compaction can, because hash compaction becomes pretty useless if its table fills up. Making a small change to a model can easily change its state space size by a factor of 2 or more, which bitstate hashing tolerates *much* better than hash compaction.

In the following sections, we explain how among currently-available tools, 3Spin and 3Murphi best support this unified approach in terms of features, performance, and ease of use.

3 Feedback

3Spin/3Murphi's most notable feature that supports our methodology is feedback after a verification run fails to find an error. These two are the only tools to report both omission analysis and recommendations, and do so in both bitstate mode and hash compaction mode.

The omission analysis uses formulas and algorithms described in previous [4] and related work [8] to compute either a probability of omitting any states or an expected number of states omitted [4]. In the latter case, the estimate is rough, but looking at the connectivity of the graph allows us to report the reliability of that result. Overall, the omission analysis helps the user understand the degree to which he can be certain the model satisfies the desired properties.

Because the configuration of the algorithms can make a big impact on accuracy, 3Spin and 3Murphi also incorporate analyses for predicting the best settings for reverifying the same (or a similar) model [4]. Pursuant to our methodology, the tools give advice on whether to follow-up with hash compaction or with bitstate hashing, along with recommended settings for each. The latest versions of these tools return the recommendations not in terms of low-level settings, but in terms of visitable state space size estimates. Perhaps the greatest benefit from this new form of recommendations is that they are not closely tied to a particular memory setting, enabling users to easily benefit from the recommendations even if they change the memory settings.

As Table 1 shows, only 3Spin and 3Murphi support this rich set of feedback features. In fact, Spin's support for recommendations in bitstate mode is derived from earlier versions of 3Spin.

4 Other Improvements or Features

This section discusses the rest of 3Spin/3Murphi's core features, following Table 1 from left to right.

Bitstate hashing. The first release of 3Spin focused on improvements to bitstate hashing that have since been integrated into Spin, starting with version 4.2.0. Prior to our work, it was believed that if memory was not terribly constrained (say 8 or more bits per state) the bitstate hashing configuration with the best accuracy was inherently slow—too slow to be more productive than iteratively using a suboptimal but fast configuration [6]. Our improvement [5,4] eliminates most of that overhead by reusing hash information in an intelligent, accuracy-preserving way.

This improvement has allowed our methodology to utilize fast *and* accurate bitstate hashing configurations, when one has a rough estimate of the state space size.

Hash compaction. Spin's implementation of hash compaction is very limited. It only supports compacted state sizes of 32 to 64 bits per state in 8 bit increments, and the size of the table must be a power of 2 (discussed in **Memory sizes** below). Both limitations inhibit Spin's ability to take advantage of available memory in minimizing the possibility of overlooking an error.

As of version 2.0, 3Spin has its own implementation of hash compaction, which has also been put in to 3Murphi. Our implementation and Murφ's support all compacted state sizes from 4 bits to 64 bits. The extent of this range is justified as follows: when fewer than about 10 bits per state are available, bitstate hashing is superior to hash compaction; 3Spin/3Murphi makes recommendations accordingly. On the high end, using a compacted state size of 64 bits is so accurate that, even if the table is almost full, the probability of *any* omissions is on the order of one in trillions. At this level of accuracy, random hardware errors are probably more likely to cause error omission than algorithmic losses.

Our implementation actually improves upon Murφ's (which is better than Spin's) in the way it determines whether a cell in the table is occupied. In addition to the memory dedicated to the compacted state, Murφ allocates a single-bit flag with each cell of the compacted table to indicate whether the cell has a state stored in it. We instead reserve the compacted state "0" to indicate that a cell is not used. As a result, we can use the bit saved to increase the compacted state size and nearly cut in half the probability of omitting an error when using the same amount of memory as Murφ.

Memory sizes. An informed choice to use a probabilistic technique is motivated by memory constraints with respect to state space size, but Spin limits the user to only power-of-2 sizes for its probabilistic data structures. 3Spin and 3Murphi allows their data structures to be of any size addressable on a 32-bit machine. Keep in mind that allocating more memory to either data structure *always* makes it more accurate—and the impact is significant. For example, when using 1024MB of memory for $k = 21$

bitstate hashing on 300 million states, the search expects to omit about 20 states. Using 3Spin with 1750MB instead leads to less than a 1% chance of omitting *any* states. If you also increase k to 35, there is less than 1 in 1000 chance of omitting any states.

Hashing. In the first release of 3Spin, we showed how to get more hash information from the hash function used by Spin, the Jenkins LOOKUP2 hash function [7], with no observable impact on coverage/accuracy. Incorporating this improvement into Spin reduced its execution time by about 25% [5] in common scenarios, because it could make just one call to Jenkins where it used to make two. 3Murphi adds support for this hash function, which can be faster than Murφ's.

Murφ uses a different hash function that enables an optimization called *differential hashing* [1]. The hash function, H_3, also has the advantage of being a *universal* hash function. Since version 3.0, 3Spin has included this hash function and similar optimizations as an alternative to Jenkins, and in many cases, the differential hashing optimization makes the universal hash function *faster* than Jenkins.

We include both hash functions as options in both tools because they are fundamentally different: Jenkins is always competitively fast but only heuristically accurate; H_3 is only heuristically fast but provably accurate.

5 Conclusion

Earlier versions of 3Spin have already made an impact by introducing features that eventually made their way into Spin itself, and here we have introduced version 3.2 of 3Spin and its new cousin 3Murphi. These tools offer a more effective verification environment to users of Spin and Murphi.

References

1. B. Cousin and J. Hélary. Performance improvement of state space exploration by regular and differential hashing functions. In *6th CAV*, pages 364–376, 1994.
2. P. C. Dillinger. 3Murphi Home Page. http://www.cc.gatech.edu/~peterd/3murphi/.
3. P. C. Dillinger. 3Spin Home Page. http://www.cc.gatech.edu/~peterd/3spin/.
4. P. C. Dillinger and P. Manolios. Bloom filters in probabilistic verification. In *Formal Methods in Computer-Aided Design*, volume 3312 of *LNCS*. Springer-Verlag, November 2004.
5. P. C. Dillinger and P. Manolios. Fast *and* accurate bitstate verification for SPIN. In *11th SPIN Workshop*, volume 2989 of *LNCS*. Springer-Verlag, April 2004.
6. G. J. Holzmann. An analysis of bitstate hashing. In *Proc. 15th Int. Conf on Protocol Specification, Testing, and Verification, INWG/IFIP*, pages 301–314. Chapman & Hall, 1995.
7. B. Jenkins. Algorithm alley: Hash functions. *Dr. Dobb's Journal*, September 1997.
8. U. Stern and D. L. Dill. Improved probabilistic verification by hash compaction. In *CHARME*, volume 987 of *LNCS*, pages 206–224. Springer-Verlag, 1995.
9. U. Stern and D. L. Dill. A new scheme for memory-efficient probabilistic verification. In *FORTE/PSTV*, pages 333–348, 1996.
10. P. Wolper and D. Leroy. Reliable hashing without collision detection. In *5th International Conference on Computer Aided Verification*, pages 59–70, 1993.

SPLAT: A Tool for Model-Checking and Dynamically-Enforcing Abstractions

Anil Madhavapeddy[1], David Scott[2], and Richard Sharp[3]

[1] Computer Laboratory, University of Cambridge
[2] Fraser Research
[3] Intel Research Cambridge
avsm2@cl.cam.ac.uk, djs@fraserresearch.org, richard.sharp@intel.com

1 Introduction

Conventional software model-checking involves (*i*) creating an abstract model of a complex application; (*ii*) validating this model against the application; and (*iii*) checking safety properties against the abstract model. To non-experts, steps (*i*) and (*ii*) are often the most daunting. Firstly how does one decide which aspects of the application to include in the abstract model? Secondly, how does one determine whether the abstraction inadvertently "hides" critical bugs? Similarly, if a counter-example is found, how does one determine whether this is a genuine bug or just a modelling artifact?

SPLAT attempts to simplify the model specification and validation tasks with a view to making model checking more accessible to regular programmers. We provide a high-level modelling language, SPL, which enables developers to specify models in terms of allowable program events (e.g. valid sequences of received network packets). We have implemented a compiler that translates SPL into both PROMELA and a number of general purpose programming languages (e.g. C, OCaml, Java). The generated PROMELA can be used with SPIN [4] in order to check static properties of the model. The generated code provides an executable model in the form of a *safety monitor*: a program which dynamically checks whether the application's behaviour deviates from the specified model. A developer can link this safety monitor against their application in order to *dynamically* ensure that the application's behaviour does not deviate from the model. If the safety monitor detects that the application has violated the model then it logs this event and terminates the application.

Although this technique simplifies model specification and validation it is, of course, not appropriate for all systems. For example, dynamically shutting down a fly-by-wire control system when a model violation is detected is not an option. However, we observe that there *are* a large class of applications where dynamic termination, while not desirable, is preferable to (say) a security breach. It is these areas in which we believe SPLAT can deliver real benefits.

Our work currently focusses on implementing servers for common Internet protocols securely and correctly. None of the major industrial implementations of protocols such as HTTP (Apache), SMTP (Sendmail/Postfix), or DNS (BIND)

P. Godefroid (Ed.): SPIN 2005, LNCS 3639, pp. 277–281, 2005.

are model-checked by their development teams. All of them regularly suffer from serious security flaws ranging from low-level buffer overflows to subtle high-level protocol errors [2]. In this paper we describe how we used SPLAT in the development of `mlssh`: a complex, high-performance SSH2-compliant server written in OCaml. Our experiences in implementing `mlssh` lead us to believe that SPLAT is accessible to regular programmers without extensive model-checking experience.

2 Discussion

To demonstrate the benefits of SPLAT we chose to use it in the development of `mlssh`: an Objective Caml [1] SSHv2 server. SSH, currently being standardized by the IETF [6], is a complex protocol combining transport-level encryption, user authentication, multiplexed data channels and remote shells. We chose to implement `mlssh` in Objective Caml, since the strong static-type safety, good UNIX syscall interface, and fast native-code output were all essential to our goals of high performance and portability.

We used the SPL language to specify sequences of network messages allowed by the SSH protocol. SPL policies, which are written using a familiar 'C'-like syntax, represent non-deterministic finite state automata. An SPL automaton's inputs are referred to as *statecalls*. In the case of `mlssh`, statecalls are generated when certain packets are received or transmitted, or some significant computation is performed by the server (e.g. deriving a shared secret via Diffie-Hellman key exchange). A simplified fragment of the `mlssh` SPL policy for the transport layer and authentication is shown in Figure 1.

Statecalls are represented by capitalized identifiers, and SPL functions use lower-case identifiers. Semicolons are used to specify sequencing (e.g. `S1; S2` specifies that state call, `S1`, must occur before state call, `S2`). Non-deterministic choice is represented by using the `either/or` construct. The `always_allow` block specifies out-of-band messages which are expected at any time but do not cause

```
automaton transport (encrypted, s_auth) {
  always_allow (Recv_ignore, Recv_debug) {
    multiple (1..) {
      either {
        Recv_kexinit; Xmit_kexinit;
        either {
          Expect_dh;      (... etc)
        } or {
          Expect_gex;     (... etc)
        }
        Recv_newkeys; Xmit_newkeys;
        encrypted = true;
      } or (encrypted && !s_auth) {
        Recv_serv_auth;
        Xmit_serv_auth_ok;
        service_auth = true;
      }
    }
  }
}
```

```
automaton auth (success, failed) {
  do {
    either {
      optional { Xmit_auth_banner; }
      either {
        Recv_auth_req_none;
        Xmit_auth_failure;
      } or
        Recv_auth_req_password;
        auth_decision (success);
      } or {
        Recv_auth_req_publickey;
        auth_decision (success);
      } or {
        Notify_auth_permanent_failure;
        failed = true;
      }
  } until (success || failed);
}
```

Fig. 1. Sample SPL fragment for an SSHv2 server

state transitions. The `multiple (1..)` block specifies that its body may occur one or more times, and `optional` allows a block to occur at most once. Although not in this example, SPL also supports a `during/handle` construct that models asynchronous message handling (particularly useful for UNIX signal handling). General recursion is prohibited, allowing us to statically allocate space for function arguments and return values. Internally, the SPLAT compiler transforms SPL into a Control Flow Automaton (CFA) [3] representation.

There are two automata specified in Figure 1: `transport` and `auth`. The `transport` automaton is parameterised over two variables: `encrypted` (representing that the channel has completed an initial key exchange and is encrypted) and `s_auth` (to indicate that the server has enabled the authentication service to the client). The `auth` automaton maintains two state variables to indicate either a successful authentication (`success`) or a permanent failure (`failed`). The `auth_decision` function call has been omitted for brevity.

Informally, the meaning of an SPL program is as follows. Each `automaton` executes in parallel and sees every statecall. If an `automaton` receives a statecall it was not expecting it reports an error. If *any* of the parallel automata report an error then the SPL model has been violated.

To make the SPL more readable, each automaton, $\mathcal{A}$, is surrounded by an implicit `always_allow` block that allows all statecalls not explicitly referenced in $\mathcal{A}$. More precisely, let $\mathcal{S}_{all}$ be the set of all statecalls referenced in the entire SPL policy. Let $\mathcal{S}(\mathcal{A})$ be the set of all statecalls referenced in the definition of automata $\mathcal{A}$. Then $\mathcal{A}$'s implicit `always_allow` block allows statecalls in the set $\mathcal{S}_{all} \setminus \mathcal{S}(\mathcal{A})$.

2.1 Executable Model

In order to actually use and enforce this model in the target application, the SPLAT compiler outputs a safety monitor designed to be linked directly against the source code of the server. The safety monitor requires that the rest of the server program cannot compromise its internal state. If this were not the case then an attacker could (say) exploit a buffer overflow to manipulate the control-flow of the safety monitor. In the case of `mlssh`, the SPLAT compiler generates OCaml code with the following interface:

```
module Automaton = struct
  exception Bad_statecall
  type statecall =
    |Xmit_ignore |Xmit_debug |Recv_kexinit |Xmit_kexinit (... etc)
  type state
  val init : unit -> state
  val tick : statecall -> state -> state
  val debug: string -> bool
end
```

This interface allows `mlssh` to initialize the safety monitor (via `Automaton.state`), and to drive it by calling `tick`. If the safety monitor ever receives

an invalid sequence of statecalls (passed via `tick` calls) then it generates the `Bad_statecall` exception, terminating the program.

Although we specifically describe an OCaml interface here, the compiler can also be easily extended to other language's type systems (e.g. Java objects), allowing server authors to write programs in their language of choice and still use the SPLAT tool-chain. In the case where languages do not make strong enough memory-safety guarantees to protect the safety monitor from the main program (e.g. C), the compiler outputs an automaton which runs in a separate UNIX process [5] and stub code which allows the server to communicate with the monitor via IPC. However, this approach is slower for more fine-grained SPL policies, as there is significantly more overhead in performing IPC than simply calling a function (as is the case with OCaml).

It is important to note that we do not enforce mechanisms for insertion of `tick` calls in server applications. In recent years, there have been a proliferation of languages being used for authoring Internet services (Python, Perl, Ruby, Erlang, Java, Eiffel, C/C++, Objective C etc.). Since different languages present such different mechanisms for writing servers, any such technique would be either too language-specific or too general to be of use. Instead, we allow SPL automata to be embedded into any of these languages and allow server authors to insert `tick`s as they see fit. In our implementation of `mlssh` we used a combination of: (*i*) manual `tick`-insertion within the server source; (*ii*) meta-programming techniques to automatically introduce `ticks` into generated code used for low-level packet parsing; and (*iii*) program slicing to automatically `tick` across API boundaries (e.g. call statecalls for every function call into the crypto library).

The executable automaton is also useful for providing high-level debugging facilities. The `debug` function in the OCaml interface above connects the automaton to a local UNIX domain socket to which a debugger process is listening, and transmits details about its internal states in real-time. Since the SPL specification typically captures higher-level information about the program's state, this complements the native language's debugging facilities with application-specific data (e.g. "what SSH packets is `mlssh` allowed to send immediately after authentication is completed?"). The automaton can also keep statistics of state transitions between program runs, aiding optimization efforts by highlighting "hotspots" in the server.

2.2 Promela Output

The SPL compiler transforms the SPL policy directly into PROMELA code, suitable for machine-checking using SPIN. In addition, message producer processes are created which continuously transmit statecalls to each automata. Model-checking this output is not very interesting beyond exhaustiveness testing to check that all the states are reachable. However, the programmer can specify additional LTL assertions which must hold for the SPL policy as a whole. These LTL assertions are checked by SPIN, and any counter-examples are translated by the compiler back into SPL line numbers (which were added to each state in the CFA by the compiler). In the code snippet shown previously, some LTL asser-

tions used are: (*i*) `encrypted` ⇒ □`encrypted` and (*ii*) `s_auth` ⇒ □`encrypted`. These indicate that when encryption is enabled, it must remain enabled for the lifetime of the session. Similarly, when the authentication service is activated, encryption must always be enabled from then on.

Although the LTL assertions are currently specified outside of the SPL specification, we are currently integrating support for the assertions into the SPL grammar, thus allowing them to be dynamically enforced in the executable automaton in the event that a model-checker is not available in the local build environment.

The LTL assertions provide the programmer with a very useful mechanism for determining whether the often informal intuitions about their server actually hold true in their SPL model. For instance, UNIX signal handlers are a common source of errors due to their extremely asynchronous nature; since they can easily be expressed in SPL via a `during/handle` clause, the programmer can include them in LTL assertions and more formally check those intuitions.

3 Tool Demonstration

In our tool demonstration, we intend to show the SPLAT tool-chain as used in the `mlssh` SSHv2 protocol server, along with a simple debugger which demonstrates the state transitions graphically in real-time. In addition, we will show how LTL assertions help the programmer enforce invariants that are currently informally mandated by the SSH specification. Finally, we hope to gather feedback from the model-checking community, as we plan to release SPLAT under a BSD-style license to encourage broader open-source adoption of model-checking techniques across multiple languages and coding styles.

References

1. X. Leroy at al. Objective Caml. `http://caml.inria.fr/`.
2. CERT Coordination Center (CERT/CC). CERT knowledgebase. `http://www.cert.org/kb/`.
3. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, George C. Necula, Gregoire Sutre, and Westley Weimer. Temporal-safety proofs for systems code. In *Proceedings of the 14th International Conference on Computer-Aided Verification*, pages pp. 526–538. Lecture Notes in Computer Science 2404, Springer-Verlag, 2002.
4. Gerard J. Holzmann. *The SPIN Model Checker : Primer and Reference Manual title*. Pearson Educational, 2003.
5. Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium*, August 2003.
6. Bill Sommerfeld. IETF Secure Shell Working Group (secsh). `http://ietf.org/html.charters/secsh-charter.html`.

Learning-Based Assume-Guarantee Verification (Tool Paper)

Dimitra Giannakopoulou[1] and Corina S. Păsăreanu[2]

[1] RIACS
[2] QSS, NASA Ames, Moffett Field, CA 94035-1000, USA
{dimitra, pcorina}@email.arc.nasa.gov

1 Introduction

Despite significant advances in the development of model checking, it remains a difficult task in the hands of experts to make it scale to the size of industrial systems. A key step in achieving scalability is to "divide-and-conquer", that is, to break up the verification of a system into smaller tasks that involve the verification of its components. Assume-guarantee reasoning [9, 11] is a widespread "divide-and-conquer" approach that uses assumptions when checking individual components of a system. Assumptions essentially encode expectations that each component has from the rest the system in order to operate correctly. Coming up with the right assumptions is typically a non-trivial manual process, which limits the applicability of this type of reasoning in practice.

Over the last few years, we have developed a collection of techniques and a supporting toolset, for performing assume-guarantee reasoning of software in an automated fashion. Our techniques are applicable both at the level of design models, and at the level of actual source code. In the heart of these techniques lies a framework that uses an off-the-shelf learning algorithm for regular languages, namely L* [1], to compute assumptions automatically.

The rest of the paper is organized as follows. Section 2 is a high-level description of our techniques for learning-based assume-guarantee reasoning of software. Section 3 discusses the tool support for our techniques and experimental results obtained from the application of our approach to some industrial size case studies, and we conclude the paper with Section 4.

2 Learning-Based Assume-Guarantee Reasoning

Analysis of Finite State Models. At the design level, our techniques target models described as labeled transition systems (LTSs) with blocking communication. We check safety properties expressed as finite state machines that describe the legal sequences of actions that a system can perform. We reason about assume-guarantee formulas $\langle A \rangle M \langle P \rangle$, where M is a component, P is a property and A is an assumption on M's environment. The formula is true if whenever M is part of a system that satisfies A, then the system must also guarantee P.

P. Godefroid (Ed.): SPIN 2005, LNCS 3639, pp. 282–287, 2005.

$1 : \langle A_1 \rangle M_1 \langle P \rangle$ $2 : \langle true \rangle M_2 \langle A_1 \rangle$ $\overline{\langle true \rangle M_1 \parallel M_2 \langle P \rangle}$	$1 : \langle A_1 \rangle M_1 \langle P \rangle$ $2 : \langle A_2 \rangle M_2 \langle P \rangle$ $3 : C(A_1, A_2, P)$ $\overline{M_1 \parallel M_2 \models P}$	$1..n : \quad \langle A_i \rangle M_i \langle P \rangle$ $n+1 : C(A_1, \cdots A_n, P)$ $\overline{\langle true \rangle M_1 \parallel \cdots \parallel M_n \langle P \rangle}$
(a)	(b)	(c)

Fig. 1. Assume-guarantee rules

Our framework is equipped with a collection of assume-guarantee rules which are sound and complete [2]. Incomplete rules can also be incorporated. The simplest (non-symmetric) assume guarantee rule (see Figure 1 (a)) establishes that property P holds for the composition of two models M_1 and M_2. In [6], we present an approach that uses this rule to perform assume-guarantee reasoning in an incremental and fully automatic fashion. The approach iterates a process based on gradually *learning* an assumption that is strong enough for M_1 to satisfy P but weak enough to be an abstraction of M_2's behavior.

The framework also handles symmetric rules [2]. These rules are instances of the rule pattern presented in Figure 1 (b). $C(A_1, A_2, P)$ represents some logical condition that involves the two assumptions and the property. For example, an instance of this rule states as the third premise that $\overline{A_1} \parallel \overline{A_2} \models P$. Here $\overline{A_1}$ and $\overline{A_2}$ denote the complement automata. Intuitively, this premise ensures that the possible common traces of M_1 and M_2, which are ruled out by the two assumptions, satisfy the property.

The approach extends to reasoning about n components. For the non-symmetric rule, we can decompose the system into two parts M_1 and $M_2' = M_2 \parallel \cdots \parallel M_n$, and apply the approach recursively for checking Premise 2. The generalization for symmetric rules follows the pattern of Figure 1 (c); its use in the context of learning-based assume-guarantee verification is illustrated in Figure 2.

The input models M_1, ..., M_n are created by the user or extracted from source code, using automated abstraction techniques, as discussed later in this section. At each iteration, L* generates approximate assumptions A_1, ... A_n. Model checking is then used to determine whether $\langle A_i \rangle M_i \langle P \rangle$ holds for each $i = 1..n$. If the result of any of these checks is false, then L* uses the returned counterexample to refine the corresponding assumption. The refinement process iterates until we obtain assumptions that are appropriate for showing that the first n premises hold. The last premise is then checked to discharge the assumptions; if it holds, then, according to the compositional rule, $M_1 \parallel \cdots \parallel M_n \models P$. Otherwise, the obtained counterexample is analyzed to see if it corresponds to a real error, or it is spurious, in which case it is used to refine the assumptions. The counterexample analysis is performed component wise.

For finite state systems, the iterative learning process terminates and it yields minimal assumptions [6]. In our experience, the generated assumptions are usually orders of magnitude smaller than the analyzed components, and the cost of learning-based assume-guarantee verification is small as compared to non-compositional model checking. This is often the case for well designed software,

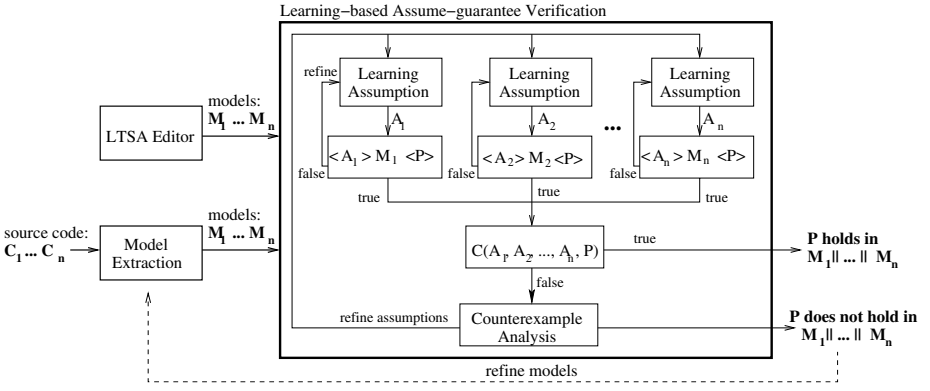


Fig. 2. Learning based assume-guarantee verification

where the interfaces between components are usually small. However, there may be cases where no single rule or no particular system decomposition yields small assumptions. Our framework partially alleviates this problem, by providing a collection of rules that the user can select and experiment with. The decomposition is still a manual process.

Analysis of Source Code. Assume first a top-down software development process, where one creates and debugs design models, which are then used to guide the development of source code (possibly in a (semi-) automatic way by code synthesis). In such a setting, the assumptions created at the design level can be used to check source code in an assume guarantee style, as presented in [8].

For cases where design models are not available, we have recently extended our framework with a component for model extraction from source code (see Figure 2). The framework is iterative: extracted models are analyzed in an assume-guarantee style, and when the analysis detects spurious errors due to the abstraction of the source code, the models are refined automatically. The extended framework advocates a clear separation between model extraction and model analysis, which facilitates the incorporation of existing well-engineered tools into it. For example, we have integrated our framework with the Magic tool that extracts finite-state models from C code using automated predicate abstraction and refinement [4]. Other tools that build finite-state models of software could also be used (e.g. Bandera for Java [7]).

3 Tool Support

Implementation. The techniques presented in the previous section have been implemented in the context of the LTSA tool [10]. The LTSA supports model checking of a system based on its architecture. It features graphical display, animation and simulation of LTSs (see Figure 3). Its input language "FSP" is

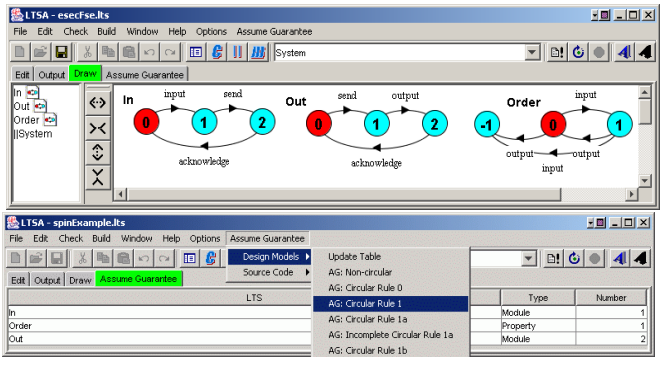


Fig. 3. LTSA GUI including Assume-Guarantee Plugin

a process-algebra style notation with LTS semantics. The LTSA has an extensible architecture which allows extra features to be added by means of plugins [5].

Our initial learning framework [6] was implemented within the core of the LTSA tool. This implementation is efficient because it can directly manipulate the internal data structures of the LTSA. However, such a solution is not sustainable, because it is hard to synchronize our code development with that of the LTSA. For this reason, we proceeded by implementing our extensions to the LTSA as the `Assume-Guarantee` plugin.

The `Assume-Guarantee` plugin extends the LTSA with a menu and a tab (see Figure 3). The menu provides options for analyzing `Design Models` and `Source Code` (uses Magic to extract models), and for selecting assume-guarantee rules. For the case of design models, all the processes in the specification are displayed in the tab, so that the user may select which components and properties participate in an assume-guarantee proof. For source code, these choices are currently hard-coded, but we intend to improve on this in the future.

Note that, at the design level, there is a significant performance overhead incurred by the plug-in implementation, which is not present in the original, non plug-in implementation. This is due to the fact that plug-ins communicate with the LTSA by placing FSP descriptions of LTSs in the Edit tab. In the future, we expect the LTSA developers to expose LTSs as objects which will enable us to do a more efficient implementation.

Experience. Within a project at NASA Ames, we have applied our techniques to the design and code of the K9 Rover Executive. The design models consist of approximately 700 lines of FSP code. The code we analyzed is about 7K lines of Java, translated from 10K lines of C++ code. Some results are shown in the tables below (monolithic refers to non-compositional verification).

We have also applied our integrated tool-set with the Magic model extractor to the verification of various safety properties of OpenSSL version 0.9.6c which has about 74,000 lines of C code. Our approach achieved two orders of magnitude space reduction when compared to Magic's non-compositional analysis [3]. Symmetric rules did consistently better than the non-symmetric one in this example.

Iteration	$\|A_i\|$	States	Transitions
1	1	294	1,548
2	2	269	1,560
3	3	**541**	**3,066**
4	5	12	69
5	6	474	2,706

Application of learning to the design of the K9 Rover Executive. Global state space: 3,630 states and 34,653 transitions. Largest state space computed by our approach: 541 states and 3,066 transitions (iteration 3). We achieve an order of magnitude space reduction.

System	States	Transitions	Memory	Time
Monolithic	183,132	425,641	952.85Mb	12m,24
Premise 1	53,215	117,756	255.96Mb	4m,49
Premise 2	13,884	20,601	118.97Mb	1m,16

Checking K9 code with JPF [12]. Use of design assumptions with the rule in Fig. 1 (a) yields a 3-fold space reduction.

4 Conclusions and Future Work

We presented a framework and its associated tool for learning-based assume-guarantee verification of software models and implementations. Our experience so far indicates that the approach has the potential of scaling to industrial size applications, especially when combined with abstraction.

Our tool is extensible: new assume-guarantee rules can be easily incorporated, and alternative tools for model extraction can be interfaced with it. Moreover, our framework is general; it relies on standard features of model checking, and could therefore be introduced in other model checking tools. For example, Magic has recently been extended to directly support learning-based assume guarantee reasoning [3]. We are also planning an implementation of our framework for the Spin model checker, in the context of a new NASA project.

Acknowledgements

We thank Howard Barringer and Jamieson Cobleigh for their contributions to our techniques and Sagar Chaki for helping with the Magic integration.

References

1. D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, Nov. 1987.
2. H. Barringer, D. Giannakopoulou, and C. S. Păsăreanu. Proof rules for automated compositional verification through learning. In *Proc. SAVCBS Workshop*, 2003.
3. S. Chaki, E. Clarke, D. Giannakopoulou, and C. Păsăreanu. Abstraction and assume-guarantee reasoning for automated software verification. *RIACS TR 05.02*, October 2004.
4. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE TSE*, 30(6):388–402, June 2004.
5. R. Chatley, S. Eisenbach, and J. Magee. Magicbeans: a Platform for Deploying Plugin Components. In *Proc. of Component Deployment (CD 2004)*.

6. J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu. Learning assumptions for compositional verification. In *Proc. of 9th TACAS*, pages 331–346, Apr. 2003.
7. J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Păsăreanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *ICSE'00*.
8. D. Giannakopoulou, C. S. Păsăreanu, and J. M. Cobleigh. Assume-guarantee verification of source code with design-level assumptions. In *Proc. of ICSE 2004*.
9. C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. on Prog. Lang. and Sys.*, 5(4):596–619, Oct. 1983.
10. J. Magee and J. Kramer. *Concurrency: State Models & Java Programs*. John Wiley & Sons, 1999.
11. A. Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and models of concurrent systems*, pages 123–144, 1985.
12. W. Visser, K. Havelund, G. Brat, and S.-J. Park. Model checking programs. In *Proc. of the Fifteenth IEEE Int. Conf. on Auto. Soft. Eng.*, pages 3–12, Sept. 2000.

Author Index