Mark D. Ryan
John-Jules Ch. Meyer
Hans-Dieter Ehrich (Eds.)

LNCS 2975

Objects, Agents, and Features

International Seminar
Dagstuhl Castle, Germany, February 2003
Revised and Invited Papers

Springer

Lecture Notes in Computer Science 2975
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Mark D. Ryan John-Jules Ch. Meyer
Hans-Dieter Ehrich (Eds.)

Objects, Agents, and Features

International Seminar
Dagstuhl Castle, Germany, February 16-21, 2003
Revised and Invited Papers

Springer

Volume Editors

Mark D. Ryan
University of Birmingham, School of Computer Science
Birmingham B15 2TT, UK
E-mail: M.D.Ryan@cs.bham.ac.uk

John-Jules Ch. Meyer
Utrecht University, Institute of Information and Computing Sciences
Intelligent Systems Group
Padualaan 14, De Uithof, P.O. Box 80.089
3508 TB Utrecht, The Netherlands
E-mail: jj@cs.uu.nl

Hans-Dieter Ehrich
Technische Universität Braunschweig, Abteilung Informationssysteme
Postfach 3329, 38023 Braunschweig, Germany
E-mail: HD.Ehrich@tu-bs.de

Library of Congress Control Number: Applied for

CR Subject Classification (1998): D.2, D.3, D.1.5, D.1.3, C.2.4, I.2.11, F.3

ISSN 0302-9743
ISBN 3-540-21989-7 Springer-Verlag Berlin Heidelberg New York

Springer-Verlag is a part of Springer Science+Business Media

springeronline.com

Printed in Germany

Typesetting: Camera-ready by author, data conversion by Olgun Computergrafik
Printed on acid-free paper SPIN: 11006299 06/3142 5 4 3 2 1 0

Preface

In recent years, concepts in object-oriented modeling and programming have been extended in several directions, giving rise to new paradigms such as agent-orientation and feature-orientation.

This volume came out of a Dagstuhl seminar exploring the relationship between the original paradigm and the two new ones. Following the success of the seminar, the idea emerged to edit a volume with contributions from participants – including those who were invited but could not come. The participants' reaction was very positive, and so we, the organizers of the seminar, felt encouraged to edit this volume. All submissions were properly refereed, resulting in the present selection of high-quality papers in between the topics of objects, agents and features. The editors got help from a number of additional reviewers, viz. Peter Ahlbrecht, Daniel Amyot, Lynne Blair, Jan Broersen, Mehdi Dastani, Virginia Dignum, Dimitar Guelev, Benjamin Hirsch, Maik Kollmann, Alice Miller, Stephan Reiff-Marganiec, Javier Vazquez-Salceda, and Gerard Vreeswijk.

Finally, we would like to take this opportunity to thank all the persons involved in the realization of the seminar and this book: attendees, authors, reviewers, and, last but not least, the staff from Schloss Dagstuhl and Springer-Verlag.

February 2004 The Editors

Table of Contents

Objects, Agents, and Features: An Introduction

John-Jules Ch. Meyer[1], Mark D. Ryan[2], and Hans-Dieter Ehrich[3]

[1] Institute of Information and Computing Sciences
Utrecht University
The Netherlands
jj@cs.uu.nl
[2] School of Computer Science
University of Birmingham
UK
M.D.Ryan@cs.bham.ac.uk
[3] Dept. of Information Systems
TU Braunschweig
Germany
HD.Ehrich@tu-bs.de

1 Introduction

There are many ways of structuring software, and the seminar focussed on an established one (object-orientation) and two emerging ones (agent-orientation and feature-orientation).

The *object* paradigm is now widely used in software technology (with programming languages like C++ and Java, and OO modelling frameworks such as UML). However, the theoretical foundations of the object paradigm are not settled yet, although clean concepts and reliable foundations are more and more demanded not only by academia but also by practitioners. In particular, the precise meaning of UML concepts is subject to wide debate.

Agents are more special kinds of objects, having more autonomy, and taking more initiative. For this reason, agent-oriented programming is sometimes referred to as 'subject-oriented' rather than 'object-oriented', indicating that an agent is much more in control of itself than an object which is manipulated by other entities (objects). There is some work on investigating typical object notions like inheritance in the context of agents. An interesting question is whether this is a fruitful way to go. Typically, agents are thought of being endowed with 'mental states' involving concepts like knowledge, belief, desires and goals, in order to display autonomous and in particular pro-active behaviour.

Features are optional extensions of functionality which may be added to a software product, in order to reflect changes in requirements. They also cut across the class structure, because implementing a feature typically involves updating several classes or objects. The more complex the system is, the harder it is to add features without breaking something; this phenomenon has been dubbed the 'feature interaction problem'. Because users like to think of a system as comprising a base system together with a number of features on top, features could potentially be seen as a structuring mechanism rivalling objects and agents.

M.D. Ryan et al. (Eds.): Objects, Agents, and Features, LNCS 2975, pp. 1–7, 2004.

The following two subsections give brief introductions into the ideas behind these paradigms.

1.1 The Agent Structuring Mechanism

Intelligent Agents. In the last decade or so the subject of 'intelligent agents' has been getting increasingly important in both the fields of software engineering and artificial intelligence [24]. (Intelligent) agents are software (or hardware) entities that display a certain degree of *autonomy* while operating in an environment (possibly inhabited by other agents) that is not completely known by the agent and typically is changing constantly. Agents possess properties like *reactiveness*, *proactiveness* and *social behaviour*, often thought of as being brought about by mental or cognitive attitudes involving knowledge, beliefs, desires, goals, intentions,..., in the literature often referred to as '*BDI attitudes*' (for beliefs, desires, intentions).

The area of agent technology covers the foundations as well as the design, implementation and application of intelligent agents, both stand alone and within (the context of) multi-agent systems. The foundations concern agent theories and in particular agent logics as a basis for agent architectures and the specification and verification of agent programs written in agent programming) languages. Design and implementation of agents involve the study of agent architectures and agent programming languages by means of which agents can be realised. Applications of agent technology are numerous (at least potentially), and range from intelligent personal assistants in various contexts to cognitive robots and e-commerce, for example [15].

Especially important is the study of *multi-agent systems*, which incorporates such issues as communication, coordination, negotiation and distributed reasoning/problem solving and task execution (e.g. distributed / cooperative planning and resource allocation). As such the area is part of the field of distributed AI. An important subfield is that of the investigation of agent *communication languages* that enable agents to communicate with other agents in a standardized and high-level manner.

Logics for Intelligent Agents. In order to describe and specify the behaviour (and the BDI attitudes, more in particular) of intelligent agents a number of logics have been proposed. The most well-known are those of Cohen and Levesque [3] and Rao and Georgeff [22]. The former is based on a linear-time temporal logic and is augmented with modal operators for belief and goals, and a possibility to express the performance of actions. On this basis Cohen and Levesque define intentions as certain (persistent) goals. In fact, the formalism is 'tiered' in the sense that it contains an 'atomic layer' describing beliefs, goals and actions of an agent, and a 'molecular layer' in which concepts like intention are defined in terms of the primitives of the atomic layer. The (title) slogan of Cohen and Levesque [3] is:

$$intention = choice + commitment$$

meaning that an intention is a particularly chosen desire or goal to which the agent is committed so that it will not drop it without a good reason (e.g. the goal is reached or the agent realises that it cannot ever be achieved any more).

The framework of Rao & Georgeff [22] is known as 'BDI logic', and is based on the branching-time temporal logic CTL*, to which modal operators for belief, desire/goal and intention are added. *A priori* these three modalities are put in the framework as independent modal operators (with their own semantic accessibility relations) but Rao & Georgeff give a systematic account of how these modalities may interact (yielding constraints on the general models for the logic). BDI logic has inspired its inventors and many others to think about the architectures of BDI agents (such as PRS - Procedural Reasoning System, [7]) and also about languages for programming agents (such as AgentSpeak(L) [21]).

Since in both of the above approaches the dynamics of BDI agents cannot be properly specified mainly due to the lack of an adequate account of the results of actions, KARO logic for describing intelligent agents was proposed, based on *dynamic logic* [9], a well-known action logic, rather than temporal logic [18, 14, 19].

Agent-Oriented Programming Languages. Of course, if our aim is to realize intelligent agents logics with which one may describe or specify them is not enough. We need means to construct these agents. A number of possible options is now available. One may directly think of architectures of agent-based systems where components of agents are given a place and more or less abstractly it is indicated how these components interact with each other. An example is the BDI architecture [22] in which beliefs, desires and intentions are realized by means of belief, goal and plan bases, and where a generic program indicates how a sense-reason-act cycle is implemented using these bases. A disadvantage of this approach is that it is both very complex and it is hard to link an agent to a logical specification by some agent logic (even a BDI logic).

Therefore other researchers have moved into the direction of building agents by means of special so-called agent-oriented programming languages, in which agent concepts are used explicitly.

The first researcher who proposed such a language was Yoav Shoham [23], the language AGENT0. Besides basic actions having to do with the particular application one has in mind (called 'private actions') such as a move action in the case of a robot, in AGENT0 one can use 'mental actions' such as updating a belief base with incoming messages, the addition of commitments (actions to be performed by the agent) and the removal of non-executable commitments, and one may also use communication actions of informing other agents and (un)requesting the performance of actions to other agents. The basic loop of the AGENT0 interpreter performs an update of the agent's mental state, that is, it reads the current messages, updates the belief and commitment bases of the agent, and consequently executes the appropriate commitments, possibly resulting in further belief updates.

Although the link with agent concepts as also used in agent logics is much more direct (via the explicit use of beliefs and commitments), in a language such

as AGENT0 there still remains a problem of directly linking agent programs with a specification in an agent logic. Some researchers have therefore proposed to use an executable agent logic, where the ideal is to directly execute a specification written in the logic that contains elements of temporal logic as well as some agent notions such as beliefs and goals. An example of this approach is the work on (Concurrent) METATEM [6]. Related is the work on the language (CON)GOLOG [17, 8] and its many variants: here programs written in that language are employed to guide a theorem prover to find a program (plan) to reach the goals of an agent. Other approaches that were proposed include the agent languages AgentSpeak(L) [21] and, inspired by this, 3APL [11, 12, 10], where it is attempted to match equipping the language with mental BDI-like concepts with giving a rigid formal semantics in which BDI-models are incorporated and which renders programs written in the language amenable to the formal analysis of correctness issues [2, 13].

1.2 The Feature Structuring Mechanism

When users evaluate competing systems and try to figure out which is the best one, they frequently do so on the basis of the *features* that the systems offer. For example, when buying a printer, one might discriminate between features on the basis of whether they have features such as:

- double-sided printing;
- ethernet compatibility;
- accepts Postscript; etc.

A given printer might or might not have each of these features.

It is useful to think of a system as comprising a 'base' system together with a set of features. The idea of the feature structuring mechanism is to describe computer systems explicitly in those terms. In its pure form, we decompose a system not in terms of objects, but in terms of a minimal *core* (which we call the base system), together with some extensions (called features).

Structuring by features is often orthogonal to structuring by objects. If we consider adding a new feature to an existing system, we find that the new feature cuts across the object structuring. Introducing a new feature does not affect just one object, but a set of them.

Features have become popular in the domain of telephony, because telephone companies have sought to gain market advantage by offering new features, and charging for them separately. Features of a telephone system include

- Call forward on busy: a call to a busy terminal is forwarded to another (pre-selected) terminal;
- Voicemail on busy: a caller to a busy terminal is offered voicemail;
- Ring back when free: a caller to a busy terminal is offered the possibility of being notified when the destination terminal becomes free.

There are hundreds of such features defined in the literature.

Features should be defined as independently as possible of the systems to which they will be added, in order to make them more abstract and to enable them to be integrated into a larger class of systems. As a consequence, when we define a feature we normally do not know what other features may be deployed with it. This means that when we do put features together, they may interact in unexpected ways. The problem of predicting and coordinating interactions between features is called the *feature interaction problem*, and receives considerable attention in the literature [5, 4, 25, 20].

Examples of feature interactions in telecommunications systems are:

- "Call Forward on Busy" and "Voice Mail on Busy": both these features try to take control of a second (incoming) call to the subscriber. This is inconsistent, so one cannot allow both features to be active on the same phone.
- The "Ring Back When Free" (RBWF) attempts to set up a call for the subscriber to a callee whose line is engaged as soon as the line becomes free. The interaction of RBWF and "Call Forward Unconditional" (CFU) leads to consistent, but potentially undesirable behaviour:
  x requests RBWF from **y**, then CFU to **z**.
  z will be notified when **y** becomes available.
  However, it was **x** who requested the notification.

Some systems have features already built-in, ready to be enabled by users, while other systems are built using an architecture which supports adding features. Examples of the latter kind include systems allowing plug-ins, like web browsers and media players, and systems allowing user-written packages like GNU Emacs and LaTeX. All of these systems are prone to feature interaction. For example, LaTeX styles may not work as intended when loaded in the wrong order, or in some cases not be compatible at all. These problems can usually be traced down to the fact that two features manipulate the same entities in the base system, and in doing so violate some underlying assumptions about these entities that the other 'features' rely on.

Feature-oriented programming is related to aspect-oriented programming, and to superimpositions. We consider each of these in turn. Aspect-oriented programming is a technique for managing program changes which cut across the existing structure. For example, AspectJ [1] provides constructs allowing the programmer to stipulate that after every call to a certain method, some other piece of code should be run. An aspect is a collection of such changes, introduced for a particular purpose. In this way, aspects could be used to implement features.

Superimpositions [16] are another technique for making cross-cutting changes to a program, though with slightly different motivation. The focus in superimposition is on adding extra code to a given program, usually to make it better behaved with respect to other concurrently running programs. In the classic example of superimposition, extra code is added to enable processes to respond to interrogations from a supervisory process about whether they are awaiting further input, and this enables smooth termination of the system.

The purpose of this volume is to explore the connections between system structuring mechanisms.

References

1. The AspectJ project. `www.aspectj.org`. Accessed January 2004.
2. R. H. Bordini, M. Fisher, C. Pardavila and M.Wooldridge. Model checking AgentSpeak. In Proceedings of the second international joint conference on autonomous agents and multiagent systems (AAMAS'03), pages 409-416, Melbourne, 2003.
3. P.R. Cohen and H.J. Levesque, Intention is Choice with Commitment, *Artificial Intelligence* 42, 1990, pp. 213–261.
4. D. Amyot and L. Logrippo, editors. *Feature Interactions in Telecommunications and Software Systems VII*. IOS Press, 2003.
5. M. Calder and E. Magill, editors. *Feature Interactions in Telecommunications and Software Systems VI*. IOS Press, 2000.
6. M. Fisher, A Survey of Concurrent METATEM – The language and Its Applications, in: *Temporal Logic – Proc. of the First Int. Conf.* (D.M. Gabbay and H.J. Ohlbach, eds.), LNAI 827, Springer, Berlin, 1994, pp. 480–505.
7. M.P. Georgeff and A.L. Lansky, Reactive Reasoning and Planning, in: Proc. 3rd Nat. Conf. on Artif. Intell. (AAAI-87), Seattle. WA, 1987, pp. 677-682.
8. G. de Giacomo, Y. Lespérance and H.J. Levesque, ConGolog, a Concurrent Programming Language Based on the Situation Calculus, *Artificial Intelligence* 121(1-2), 2000, pp. 109–169.
9. D. Harel, Dynamic Logic, in: D. Gabbay and F. Guenthner (eds.), *Handbook of Philosophical Logic, Vol. II*, Reidel, Dordrecht/Boston, 1984, pp. 497–604.
10. K.V. Hindriks, Agent Programming Languages: Programming with Mental Models, PhD. Thesis, Utrecht University, 2001.
11. K.V. Hindriks, F.S. de Boer, W. van der Hoek and J.-J. Ch. Meyer, Formal Semantics for an Abstract Agent Programming Language, in: *Intelligent Agents IV* (M.P. Singh, A. Rao and M.J. Wooldridge, eds.), LNAI 1365, Springer, 1998, pp. 215–229.
12. K.V. Hindriks, F.S. de Boer, W. van der Hoek and J.-J. Ch. Meyer, Agent Programming in 3APL, *Int. J. of Autonomous Agents and Multi-Agent Systems* 2(4), 1999, pp.357–401.
13. K.V. Hindriks, F.S. de Boer, W. van der Hoek and J.-J. Ch. Meyer, A Programming Logic for Part of the Agent Language 3APL, in: (Proc. First Goddard Workshop on) Formal Approaches to Agent-Based Systems (FAABS 2000) (Rash, J.L., Rouff, C.A., Truszkowski, W., Gordon, D. and Hinchey, M.G. eds.),, LNAI 1871, Springer, Berlin/Heidelberg, 2001, pp. 78-89.
14. W. van der Hoek, B. van Linder and J.-J. Ch. Meyer, An Integrated Modal Approach to Rational Agents, in: *Foundations of Rational Agency* (M. Wooldridge and A. Rao, eds.), Applied Logic Series 14, Kluwer, Dordrecht, 1998, pp. 133–168.
15. N.R. Jennings and M.J. Wooldridge, *Agent technology: Foundations, Applications, and Markets*, Springer, Berlin, 1997.
16. S. Katz. A superimposition control construct for distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(2):337–356, April 1993.
17. H.J. Levesque, R. Reiter, Y. Lespérance, F. Lin and R.B. Scherl, GOLOG: A Logic Programming Language for Dynamic Domains, J. of Logic Programming 31, 1997, pp. 59–84.

18. B. van Linder, Modal Logics for Rational agents, PhD. Thesis, Utrecht University, 1996.
19. J.-J. Ch. Meyer, W. van der Hoek and B. van Linder, A Logical Approach to the Dynamics of Commitments, *Artificial Intelligence* 113, 1999, 1–40.
20. M. C. Plath and M. D. Ryan. Feature integration using a feature construct. *Science of Computer Programming*, 2001.
21. A.S. Rao, AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language, in: *Agents Breaking Away* (W. van der Velde and J. Perram, eds.), LNAI 1038, Springer, Berlin, 1996, pp. 42–55.
22. A.S. Rao and M.P. Georgeff, Modeling Rational Agents within a BDI-Architecture, in *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning (KR'91)* (J. Allen, R. Fikes and E. Sandewall, eds.), Morgan Kaufmann, 1991, pp. 473–484.
23. Y. Shoham, Agent-Oriented Programming, *Artificial Intelligence* 60(1), 1993, pp. 51–92.
24. M.J. Wooldridge and N.R. Jennings (eds.), *Intelligent Agents*, Springer, Berlin, 1995.
25. P. Zave, FAQ Sheet on Feature Interaction. At `www.research.att.com/~pamela/faq.html`, accessed January 2004.

Coordinating Agents in OO

Frank S. de Boer[1,2,3], Cees Pierik[1],
Rogier M. van Eijk[1], and John-Jules Ch. Meyer[1]

[1] Institute of Information and Computing Sciences, Utrecht University, The Netherlands
{frankb,cees,rogier,jj}@cs.uu.nl
[2] CWI, Amsterdam, The Netherlands
[3] LIACS, Leiden, The Netherlands

Abstract. In this paper we introduce an object-oriented coordination language for multi-agents systems. The beliefs and reasoning capabilities of an agent are specified in terms of a corresponding abstract data type. Agents interact via an extension of the usual object-oriented message passing mechanism. This extension provides the autonomy that is required of agents but which objects in most object-oriented languages do not have. It consists of an explicit answer statement by means of which an agent can specify that it is willing to accept some specified messages. For our coordination language we also present a formal method for proving correctness. The method extends and generalizes existing assertional proof methods for object-oriented languages.

1 Introduction

Software agents and multi-agent systems have become a major area of research and development activity in computing science, with a huge potential of applications in many different domains. One may even speak of an emerging new paradigm of *agent-oriented programming* in which the concept of an agent as a software entity displaying autonomous behavior based on a private mental state and communicating and co-operating with other agents in a common environment plays a central role.

However, it is also clear that the very concept of agenthood is a complex one, and it is imperative that programming agents be amenable to precise and formal specification and verification, at least for some critical applications. This is recognized by (potential) appliers of agent technology such as NASA, which organizes specialized workshops on the subject of formal specification and verification of agents [1, 2]. Since agents comprise a new computational concept, tools for verification are not immediately available, and it is not clear *a priori* how these can be obtained. (There *are* special (modal) logics for describing agents but generally it is not clear how these are to be related to concrete agent programs. This is an instance of what is generally called the gap between agent theories and agent implementation [3, 4].)

A possibly viable way to go is to consider the relation of agents with the much more established paradigm of object-oriented programming. Sometimes agents

M.D. Ryan et al. (Eds.): Objects, Agents, and Features, LNCS 2975, pp. 8–25, 2004.

are compared to objects as being special kinds (classes) of objects where the main distinction is the autonomous aspect of their behavior. Typically, agents cannot be just commanded to execute some action (method), but they should be requested to perform some action on behalf of another agent. An agent that is requested to do such will consider whether it is in line with its own mental state (beliefs, desires, goals, intentions etc.) to do so (and may refuse) [3].

If a relationship with OO programming can be established this also opens up a way to use specification and verification methods and techniques from the realm of OO. In the present paper we investigate this line of research by considering a simple agent programming language (in the style of [5]) that is an 'agentified' adaptation of an object-oriented language for which a correctness logic (and associated tools) is known. Of course, special attention has to be given to the agent-oriented features such as agent communication and coordination, which results in a novel Hoare-style correctness logic.

This novel Hoare logic constitutes a promising first step towards realizing C.A.R. Hoare's Grand Challenge for computer science in the context of agent-oriented programming. In general, information technology industry is still in need for more reliable techniques that guarantee the quality of software. This need is recognized by Hoare as a Grand Challenge in his proposal of the verifying compiler [6].

A verifying compiler resembles a type checker in the sense that it *statically* checks properties of a program. The properties that should be checked have to be stated by the programmer in terms of *assertions* in the code. The first sketch of such a tool is given by Floyd [7]. In recent years, several programming languages (e.g., EIFFEL [8]) have been extended to support the inclusion of assertions in the code. These assertions can be used for testing but they can also be used as a basis for a *proof outline* of a program. Hoare logic [9] can be seen as a systematic way of generating the verification conditions which ensure that an annotated program indeed constitutes a proof-outline. A verifying compiler then consists of a front-end tool to a theorem prover which checks the automatically generated verification conditions (see, for example, [10]).

A fundamental idea underlying Hoare logics as introduced by C.A.R. Hoare is that it allows the specification and verification of a program at the same level of abstraction as the programming language itself. This idea has some important consequences for the specification of properties of agent-oriented programs that involve dynamically allocated variables for dynamic referencing of agents. Firstly, this means that, in general, there is no explicit reference in the assertion language to the heap (using object-oriented jargon) which stores the dynamically allocated variables at run-time. Moreover, it is only possible to refer to variables in the heap that do exist. Variables that do not (yet) exist never play a role. In general, these restrictions imply that first-order logic is too weak to express some interesting properties of dynamically allocated heap structures.

The main contribution of this paper is a Hoare logic for reasoning about an object-oriented programming language for the coordination of intelligent agents. The Hoare logic is based on an assertion language which allows the specification

of run-time properties at an abstraction level that coincides with that of the programming language. The coordination language itself supports an abstract data type for the beliefs and reasoning capabilities of an agent. This abstract data type allows the integration of heterogenous agents which are defined by different implementations of their beliefs and reasoning capabilities. Agents can be created dynamically and to ensure autonomy, agents can interact only via method calls, that is, they cannot access directly each other beliefs (which are thus encapsulated). Agents however can communicate their beliefs via the parameter passing mechanism of method calls. The autonomy of agents is additionally supported by means of answer statements which specify the methods an agent is willing to answer.

This paper is organized in a straightforward way. In Sect. 2 we introduce the programming language. In the following section we describe the assertion language that is used to annotate programs. Section 4 contains the Hoare logic. This paper ends with conclusions and some hints for future work.

2 The Programming Language

An agent system consists of a dynamically evolving set of independently operating agents that interact with each other through requests. Each agent belongs to an agent class that specifies the requests that agents of the class can be asked to answer. Agents have explicit control over the invocations of its methods: there is a special language construct that is used to specify at which point in its execution an agent is willing to answer which requests (although not from which agent).

Furthermore, an agent has a belief base and a program that governs its behavior. This program is built from traditional programming statements like sequential composition, choice and while-loops and contains atomic operations for handling question invocations, for querying and maintaining the belief base and an operation for integrating new agents into the system. Each agent has a private set of instance variables to keep track of and refer to the agents in the system that it knows (and thus can ask questions to).

Agents maintain their own subjective belief base. We will assume that a belief base is an element of an abstract data type Belief. In this paper $p, q, \ldots$ will denote typical values of this type. Beliefs can appear in several ways in the program. For example, an agent can pass beliefs to other agents by sending them as arguments in requests. The receiving agents store the incoming beliefs in the formal parameters of the corresponding methods. A parameter that refers to a formula will have type Belief.

An agent can perform three operations on beliefs besides passing beliefs to other agents. It can add a belief to its belief base by means of the statement assert(p) and retract it from its beliefs by the command retract(p). Finally, it can test whether a belief p follows from its belief base by the command query(p). The result of a query is always a boolean value that can be assigned to a (boolean) variable. We do not allow assignments of beliefs to instance variables because

the beliefs of an agent are stored in a separate belief base and we do not want to mix up the belief base of an agent and the values of its internal state.

We will now formally define the expressions, statements, method declarations, classes and programs of the programming language. We start with the expressions. The expressions in the language are built from local variables and instance variables and the keywords self and null that are typical for the object-oriented paradigm. In the following definition, u denotes an arbitrary local variable, and x is an instance variable of an agent. Agents only have direct access to their own belief base and their own instance and local variables.

Definition 1 (Expressions).

$$e \in \text{Expr} \ ::= \ \mathsf{null} \mid \mathsf{self} \mid u \mid \mathsf{self}.x \mid e_1 = e_2 \mid p$$

The keyword self always refers to the agent that is actively executing the method. The keyword null denotes the null reference.

We assume a set of agent class names $\mathcal{A}$ with typical element a. The set of types $\mathcal{T}$ in the program is the set $\mathcal{A} \cup \{\mathsf{bool}, \mathsf{Belief}\}$. The language is strongly-typed. Only formal parameters of a method can have type Belief. We will silently assume that all expressions and statements are well-typed.

An important feature of the proposed language is that each agent has its own internal activity that is executed in parallel with the activities of the other agents. The internal activity is described by a (sequential) statement S. We allow the following statements.

Definition 2 (Statements).

$$\begin{aligned} S \in \text{Stat} ::= {} & \mathsf{skip} \mid u := e \mid \mathsf{self}.x := e \mid S_1 \ ; \ S_2 \mid \mathsf{if}\ e\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi} \\ & \mid \mathsf{while}\ e\ \mathsf{do}\ S\ \mathsf{od} \mid u := \mathsf{new}(a) \mid u := e_0.m(e_1, \ldots, e_n) \\ & \mid \mathsf{answer}(m_1, \ldots, m_n) \mid \mathsf{assert}(e) \mid \mathsf{retract}(e) \mid u := \mathsf{query}(e) \end{aligned}$$

Observe that an agent can only alter the value of its own instance variables (by means of assignments of the form $\mathsf{self}.x := e$). Thus each agent has control over its own instance variables. A statement like $u := \mathsf{new}(a)$ denotes the creation of a new agent, and the storage of a reference to this agent in the local variable u. Creation of an agent also implies that the internal activity of an agent is activated.

A statement $u := e_0.m(e_1, \ldots, e_n)$ involves a request to agent e_0 to execute method m with actual parameters $e_1, \ldots, e_n$. Such a request will only be fulfilled if the receiving agent e_0 indicates by means of a statement $\mathsf{answer}(m_1, \ldots, m_n)$ that it is willing to answer a request by executing a message out of the list of methods $m_1, \ldots, m_n$. If $m \in \{m_1, \ldots m_n\}$ then m might be selected for execution. Execution of the requesting agent is suspended until its request is answered. When executing a statement $\mathsf{answer}(m_1, \ldots, m_n)$, an agent will be suspended until it receives a request to execute one of the methods $m_1, \ldots, m_n$. If some requests are already pending at execution of $\mathsf{answer}(m_1, \ldots, m_n)$ then one of these is chosen non-deterministically. The answering agent executes the method and returns a value. After that, both agents resume their own internal activity.

A method declaration specifies the name of the method m, a sequence of formal parameters $\bar{u} = u_1, \ldots, u_n$, a body S and a return value e.

Definition 3 (Method declarations).

$$\mathrm{md} \in \mathrm{Meth} ::= m(u_1, \ldots, u_n)\{\ S\ \texttt{return}\ e\ \}$$

Formal parameters can have type Belief. In that case, the actual parameter should be a belief. We do not allow assignments to formal parameters. For technical convenience, we also assume that the body S of a method contains no answer statements. Thus an agent always answers one request at a time. However, it may delegate (parts of) its task to other agents by sending requests in the body of a method.

An agent is an instance of an agent class A.

Definition 4 (Agent classes).

$$A \in \mathrm{Agent} ::= (a, X, M, S)$$

Each agent class consists of a unique name a, a set of typed instance variables X, a set of method declarations M and a statement S. The statement S specifies the internal activity of the agent. The belief base of an agent is initially empty (i.e., equivalent to true).

Definition 5 (Multi-agent programs). *A multi-agent program π is a tuple $(S, A_1, \ldots, A_k)$ of an initial statement S and agent classes A_1 to A_k.*

The initial statement S starts the execution of the program. It typically creates some agents and then terminates. The keyword self is illegal in S because the command is not linked to a particular agent. For the same reason, it can neither contain answer statements nor operations on the belief base.

In order to formalize the semantics of a program π we introduce for each agent name a of a class defined in π an infinite set of agent identities Id_a. A global state σ of π is a family of *partial* functions

$$\sigma_a \in \mathsf{Id}_a \to F,$$

where F denotes the set of assignments of values to the instance variables of class a, a being a name of a class defined by π. We assume an implicit auxiliary variable belief which represents the belief base, that is, $\sigma_a(i)(\mathsf{belief})$ represents the belief base of the agent i. It should be observed here that the auxiliary variable belief does not appear in programs and can be accessed only via the operations of the abstract data type of beliefs. An agent $i \in \mathsf{Id}_a$ *exists* in σ if $\sigma_a(i)$ is defined, in which case $\sigma_a(i)(x)$ gives the value of the instance variable x of i.

A local configuration is a pair $\langle \tau, S \rangle$, where S denotes the statement to be executed, and where τ is an assignment of values to the local variables of π. Additionally we assume that self is a local variable, that is, $\tau(\mathsf{self})$ denotes the agent that is executing the statement S.

In order to model the (recursive) execution of answer statements we introduce a stack of method invocations, denoted by γ, which consists of a finite sequence of local configurations $\langle \tau_1, S_1 \rangle \cdots \langle \tau_n, S_n \rangle$, where the executing agent is given by $\tau_i(\mathsf{self})$, i.e., $\tau_i(\mathsf{self}) = \tau_j(\mathsf{self})$, for $1 \leq i, j \leq n$. Moreover, every local configuration $\langle \tau_{i+1}, S_{i+1} \rangle$, $1 \leq i < n$, describes the execution of a method which is generated by the execution of a corresponding answer statement $\mathsf{answer}(\ldots, m, \ldots); S_i$ in the preceding local configuration.

In the sequel, by $\gamma \cdot \langle \tau, S \rangle$ we denote the result of appending the local configuration $\langle \tau, S \rangle$ to the stack γ.

A global configuration of π is a pair (σ, T), where T is a set of stacks (one for every existing agent in σ). The operational semantics then can be defined in a fairly straightforward manner in terms of a transition relation on global configurations. The semantics is parametric in the interpretation of the operations of the abstract data type of beliefs. That is, we assume semantic functions

$$[\![\mathsf{assert}(e)]\!](\sigma, \tau)$$

and

$$[\![\mathsf{retract}(e)]\!](\sigma, \tau)$$

which specify the global state σ' resulting from a corresponding revision of the belief base $\sigma(\tau(\mathsf{self}))(\mathsf{belief})$. The semantic function $[\![\mathsf{query}(e)]\!](\sigma, \tau)$ does not generate any side-effect and encodes whether the belief denotes by e holds (in the belief base of $\tau(\mathsf{self})$.

We have the following transition for the assert statement (the retract statement is described in a similar manner).

Assert. Let $\mathsf{assert}(e)(\sigma, \tau) = \sigma'$ in the following transition.

$$(\sigma, T \cup \{\gamma \cdot \langle \tau, \mathsf{assert}(e); S \rangle\}) \rightarrow (\sigma', T \cup \{\gamma \cdot \langle \tau, S \rangle\}) \ .$$

This transition singles out one agent and its corresponding execution stack

$$\gamma \cdot \langle \tau, \mathsf{assert}(e); S \rangle$$

where the top statement starts with an assert statement. The execution of this statement simply consists of updating the global state as specified by the semantics of the assert statement and of a corresponding update of the statement still to be executed. Note that all the other agents in T remain the same, i.e., we describe concurrency by interleaving.

We have the following main cases describing the semantics of method calls.

Method Invocation. In the following transition the stack of the caller is given by

$$\gamma \cdot \langle \tau, u := e_0.m(\bar{e}); S \rangle.$$

The stack of the callee is given by

$$\gamma' \cdot \langle \tau', \mathsf{answer}(\ldots, m, \ldots); S' \rangle.$$

Furthermore, the agent $\tau'(\mathsf{self})$ indeed is addressed by e_0 in the call $e_0.m(\bar{e})$, i.e., the agent identity $\tau'(\mathsf{self})$ equals the value of the callee denoted by the expression e_0 evaluated in σ and τ.

$$(\sigma, T \cup \{\gamma \cdot \langle \tau, u := e_0.m(\bar{e}); S\rangle, \gamma' \cdot \langle \tau', \mathsf{answer}(\ldots, m, \ldots); S'\rangle) \rightarrow \\ (\sigma, T \cup \{\gamma \cdot \langle \tau, \mathsf{wait}(u); S\rangle, \gamma' \cdot \langle \tau', S'\rangle \cdot \langle \tau'', S''\,\mathtt{return}\ e\rangle)$$

The stack of the callee is extended with a new local configuration

$$\langle \tau'', S''\ \mathtt{return}\ e\rangle\ ,$$

where $S''\ \mathtt{return}\ e$ is the body of method m. The local state τ'' assigns to the formal parameters of m the values of the corresponding actual parameters $\bar{e}$ (evaluated in σ and τ) and $\tau''(\mathsf{self}) = \tau'(\mathsf{self})$. In order to model the return we additionally assume that a local variable `return` is set in τ'' to the caller $\tau(\mathsf{self})$. The waiting of the caller for the return value is modelled by the auxiliary statement $\mathsf{wait}(u)$. The semantics of this wait statement is described in the following transition which describes the return from a method call.

Method Return. In the following transition the stack of the suspended caller is given by

$$\gamma \cdot \langle \tau, \mathsf{wait}(u); S\rangle\ .$$

The stack of the terminated callee is given by

$$\gamma' \cdot \langle \tau', S'\rangle \cdot \langle \tau'', \mathtt{return}\ e\rangle\ .$$

Furthermore, we require that the callee indeed returns a call invoked by the agent $\tau(\mathsf{self})$, i.e., $\tau(\mathsf{self}) = \tau''(\mathtt{return})$. Finally, in the following transition v is the value of return expression e (evaluated in σ and τ'') and the result of assigning v to u is denoted by $\tau\{u := v\}$.

$$(\sigma, T \cup \{\gamma \cdot \langle \tau, \mathsf{wait}(u); S\rangle, \gamma' \cdot \langle \tau', S'\rangle \cdot \langle \tau'', \mathtt{return}\ e\rangle) \rightarrow \\ (\sigma, T \cup \{\gamma \cdot \langle \tau\{u := v\}, S\rangle, \gamma' \cdot \langle \tau', S'\rangle)$$

Note that after the return the stack of the callee is simply popped.

The transitions for the other statements are standard and therefore omitted.

3 The Assertion Language

The proof system that we introduce in the next section is tailored to a specific assertion language. We introduce the syntax and semantics of the assertion language in this section. Between the lines we also explain some of the design decisions behind the language.

An important design decision is to keep its abstraction level as close as possible to the programming language. In other words, we refrain as much as possible from introducing constructs that do not occur in the programming language. This should make it easier for programmers to annotate their programs.

The assertion language is strongly-typed similar to the programming language. By $[\![l]\!]$ we denote the type of expression l. The type of self is determined by its context. We will silently assume that all expressions are well-typed.

The set of logical expressions is defined by the following grammar.

Definition 6 (Logical expressions).

$$\begin{aligned} l \in \mathrm{LExpr} ::= {} & \mathsf{null} \mid \mathsf{self} \mid u \mid z \mid l.x \mid l.\phi \mid l_1 = l_2 \\ & \mid \mathsf{if}\ l_1\ \mathsf{then}\ l_2\ \mathsf{else}\ l_3\ \mathsf{fi} \mid l[l'] \mid l.\texttt{length} \end{aligned}$$

Here ϕ denotes either a belief p, a local variable u or a logical variable z (of type Belief*).*

The expressions in the programming language are the basis for the set of logical expressions. The variable z is a logical variable. Logical variables are auxiliary variables that do not occur in the program. Therefore we are always sure that the value of a logical variable can never be changed by a statement. Logical variables are used to express the constancy of certain expressions (for example in a proof rule for message passing below). Logical variables also serve as bound variables for quantifiers.

In a proof outline one can express assumptions about the values of instance variables of an agent (by expressions of the form $l.x$) and about its beliefs (by means of $l.\phi$). These two types of statements have a different meaning. An expression $l.\phi$ is true if the belief denoted by ϕ follows from agent l's belief base, whereas $l.x$ simply refers to the value of instance variable x of agent l.

Example 1. Let u be a formal parameter of a method, and let its type be Belief. Then the assertion $\mathsf{self}.u = \mathsf{self}.x$ says that the boolean instance variable x of agent self signals whether belief u currently follows from its belief base. On the other hand, the assertion $u = \mathsf{self}.x$ is not well-typed because an instance variable cannot have type Belief.

We included logical expressions of the form $\mathsf{if}\ l_1\ \mathsf{then}\ l_2\ \mathsf{else}\ l_3\ \mathsf{fi}$ to be able to reason about aliases (see Sect. 4.1). Logical variables can additionally have type t^*, for some t in the set of types $\mathcal{T}$ extended with the type of integers. This means that its value is a *finite sequence* of elements from the domain of t. This addition adds expressive power to the language needed to express certain properties of dynamically changing multi-agent systems. We use $l[l']$ to select the element at position l' in the sequence l. The length of a sequence l is denoted by $l.\texttt{length}$.

Formulas are built from expressions in the usual way.

Definition 7 (Formulas).

$$P, Q \in \mathrm{Ass} ::= l_1 = l_2 \mid \neg P \mid P \wedge Q \mid \exists z : t(P)$$

Note that as atomic formulas we only allow equations. Furthermore we have the usual boolean operations of negation and conjunction (using the standard abbreviations for the other boolean operations). The type t in the (existential)

quantification $\exists z : t(P)$ can be of any type. For example, t can be a or a^*, for some class a. In the first case, the formula means that P holds for an *existing* agent of class a. In the latter case, P should hold for a sequence of such agents. We sometimes omit the type information in $\exists z : t(P)$ if it is clear from the context. The standard abbreviations like $\forall z P$ for $\neg\exists z \neg P$ are valid.

Example 2 (Agreement). Many interesting properties can be expressed in the assertion language. We can, for example, say that all agents of an agent class a share the opinion of a particular agent on the belief denoted by ϕ. This is expressed by the formula $\forall z : a(\mathsf{self}.\phi = z.\phi)$.

Example 3 (Conflicting agents). We also allow quantification over beliefs in the assertion language. Thus one can, for example, express that two agents u and v disagree about everything by the formula $\forall z : \mathsf{Belief}(u.z = \neg(v.z))$.

Assertion languages for dynamically allocated variables inevitably contain expressions like $l.x$ that are normally undefined if, for example, l is null. However, as an assertion their value should be defined. We solved this problem by giving such expression the same value as null. By only allowing the non-strict equality operator as a basic formula, we nevertheless ensure that formulas are two-valued. For example, the expression $u.x = \mathsf{true}$ indicates that the value of the (boolean) instance variable x of object u is true *and* that u does not equal null. On the other hand, $u.x = \mathsf{null}$ signals that either u equals null or that the value of field x of object u equals null. If we omit the equality operator then the value of a boolean logical expression l is implicitly compared to true.

Example 4 (Finite sequence). The following formula

$$\exists z : a^* \forall n(1 \leq n \wedge n < z.\mathsf{length} \rightarrow \forall z' : \mathsf{Belief}(z[n].z' \rightarrow z[n+1].z'))$$

expresses that there exists a finite sequence of agents of class a which is monotonically increasing with respect to their beliefs.

Formally the semantics of assertions is defined along the lines of [11] with respect to a local state which specifies the values of the local variables and the value of self, a global state which specifies the values of the instance variables and the belief bases of the existing agents, and, finally, a logical environment ω which specifies the values of the logical variables. The semantics is parametric in the interpretation

$$[\![\mathsf{query}(p)]\!](\sigma, \tau)$$

of the query operation of the abstract data type Belief, which, as discussed already above, encodes if the belief p holds in the belief base $\sigma(\tau(\mathsf{self}))(\mathsf{belief})$. For example, we define the boolean value

$$[\![l.\phi]\!](\omega, \sigma, \tau)$$

resulting from the evaluation of $l.\phi$, inductively by

$$[\![\mathsf{query}(p')]\!](\sigma, \tau\{\mathsf{self} := [\![l]\!](\omega, \sigma, \tau)\}),$$

where $p' = [\![\phi]\!](\omega, \sigma, \tau)$. This latter semantic function is given by the following three clauses.

- $[\![p]\!](\omega, \sigma, \tau) = p$
- $[\![u]\!](\omega, \sigma, \tau) = \tau(u)$ (u is a local variable of type Belief)
- $[\![z]\!](\omega, \sigma, \tau) = \omega(z)$, ($z$ is a logical variable of type Belief)

4 Proof Outlines

Proof outlines as a means for representing a correctness proof for shared-variable concurrency were introduced by S. Owicki and D. Gries [12]. A proof outline is a program annotated with formulas (assertions) that describe the program state when execution reaches that point. For simplicity, we will assume that every statement occurs between two assertions (its precondition and postcondition). The precondition of a statement describes the state upon arrival at the statement, whereas the postcondition describes the state on completion of the statement. Usually, the postcondition of a statement will be the precondition of the next statement.

Correctness of a proof outline is checked in three stages. The first stage analyzes the *local* correctness of the proof outline. Local correctness comprises checking the correctness of local statements that is, those statements which do *not* involve communication via method calls. Roughly, local correctness amounts to proving that the precondition implies the postcondition of the statement. For example, for a simple assignment to a local variable $u := e$ with precondition P and postcondition Q this means checking if $P \rightarrow Q[e/u]$ holds. By $[e/u]$ we denote the substitution of u by e. It is defined by induction on the structure of assertions. The formula $P \rightarrow Q[e/u]$ is called the verification condition of the statement. In the Sects. 4.1-4.2 we investigate the local correctness of the special commands in our language.

The second stage of the proof method consists of a variant of the cooperation test (see, e.g., [13]) which deals with the communication statements (i.e., method calls and answer statements). This test thus concerns the communication of the agents. The cooperation test inspects if all possible matches of a request and answer satisfy their specification. It is described in Sect. 4.3.

The final stage of the system is an interference freedom test. It tests if the validity of the proof outline in a particular agent class is not invalidated by the execution of commands by other agents (possibly of the same class). The interference freedom test that we propose is described in Sect. 4.4. It involves an adaptation of the interference freedom test in [12] to our object-oriented agent language.

4.1 Agent Creation and Instance Variables

The typical object-oriented programming statements can be proved locally correct by means of the Hoare logic described in [11]. (We also take subtyping, and inheritance into account in [11], but that does not invalidate the verification conditions. It only means they can be further simplified.)

From that paper, we use the substitutions $[e'/e.x]$ and $[\mathsf{new}(a)/u]$. The first subsitution corresponds to the assignment $e.x := e'$, where x is an instance variable of agent e. The subsitution $[\mathsf{new}(a)/u]$ corresponds to the statement $u := \mathsf{new}(a)$. Despite their appearance these operations do not correspond to the usual notion of subsitution. For example, the substitution $[e'/e.x]$ has to take possible aliases of $e.x$ into account, e.g., logical expressions of the form $l.x$, where l *after* the assignment $e.x := e'$ denotes the same agent as e, and thus should also be replaced by e'. Since we cannot identify such aliases statically the substitution operation $[e'/e.x]$ replaces $l.x$ by the conditional expression $\mathsf{if}\ l[e'/e.x] = \mathsf{self}\ \mathsf{then}\ e'\ \mathsf{else}\ l[e'/e.x].x\ \mathsf{fi}$.

Since we only allow assignments to instance variables of the form $\mathsf{self}.x := e$ we can instantiate the substitution $[e'/e.x]$ to $[e/\mathsf{self}.x]$. For an assignment $\mathsf{self}.x := e$ with precondition P and postcondition Q we obtain the verification condition $P \to Q[e/\mathsf{self}.x]$.

Agent creation is slightly more complex because first of all in the state before its creation the new agent does not exist and so we cannot refer to it! This problem however can be solved by a static analysis of ocurrences of u (which denotes the newly created agent after its creation) and observing that u can only be compared with other variables and can be dereferenced. These different contexts can be resolved statically, e.g. $(u = v)[\mathsf{new}(a)/u]$ results in false (for further details we again refer to [11]). Furthermore, the new agent will start executing its internal activity after its creation. We therefore also must ensure that the precondition of the internal activity of the new agent is satisfied by the state that results from its creation. Let P be the precondition of $u := \mathsf{new}(a)$ and let Q be its postcondition. Let P' be the precondition of the internal activity of agents of class a. The corresponding verification condition is the formula

$$P \to (Q[\mathsf{new}(a)/u] \wedge P'[u/\mathsf{self}][\mathsf{new}(a)/u]) \ .$$

The subsitution $[u/\mathsf{self}]$ simply replaces every occurrence of self in P' by u. This yields the desired result because the new agent is known in the context of its creation by the reference u.

4.2 Reasoning about Beliefs and Belief Updates

Our assertion language allows one to express assumptions about what follows from the belief base of a particular agent by means of expressions of the form $l.\phi$. Such an expression is true if ϕ follows from the belief base of agent l. It is clear that belief revision commands influence the validity of such expressions. We therefore have to explore the logical foundations of such belief updates.

In the context of this paper we will restrict our attention to a simple setting where, given some signature of predicate and function symbols, beliefs are (ground) literals. In other words, we now instantiate the abstract data type Belief with a concrete data type. Reasoning about more advanced belief data types is left for future research.

We first consider the command $\mathsf{assert}(e)$. After execution of this command we know that $\mathsf{self}.e$ holds. A first attempt to formulate a verification condition

would be to check whether the postcondition is implied by the precondition if we replace all occurrences of $\mathsf{self}.e$ in the postcondition by true. However, there might also be other expressions that denote the same agent as self. In other words, we are faced with the aliasing problem. Moreover, the belief e may also have aliases. However, we can solve this problem by introducing a conditional expression that takes this possibility into account. By $[\mathsf{true}/\mathsf{self}.e]$ we denote the substitution of all aliases of $\mathsf{self}.e$ by true. It is defined as follows on a formula $l.\phi$.

$$l.\phi[\mathsf{true}/\mathsf{self}.e] \equiv \mathsf{if}\ l = \mathsf{self} \wedge \phi = e\ \mathsf{then\ true\ else}\ l.\phi\ \mathsf{fi}$$

On all other formulas the substitution corresponds to the usual notion of structural substitution. For a specification $\{P\}\ \mathsf{assert}(e)\ \{Q\}$ the verification condition is then given by $P \to Q[\mathsf{true}/\mathsf{self}.e]$.

Example 5. Let us consider the action $\mathsf{assert}(p)$. Suppose we want to prove that after this command the postcondition $\forall z : \mathsf{Member}(\mathsf{self}.p = z.p)$ holds. That is, we want to ensure that after adding p to our belief base all agents of sort Member share our belief of p. The weakest precondition of this command given the postcondition is

$$\forall z : \mathsf{Member}(\mathsf{self}.p = z.p)[\mathsf{true}/\mathsf{self}.p]\ ,$$

which amounts to checking that for every agent z in class Member we have

$$(\mathsf{if}\ \mathsf{self} = \mathsf{self} \wedge p = p\ \mathsf{then\ true\ else}\ \mathsf{self}.p\ \mathsf{fi} = \mathsf{if}\ z = \mathsf{self} \wedge p = p\ \mathsf{then\ true\ else}\ z.p\ \mathsf{fi})\ .$$

The latter formula is equivalent to $\forall z : \mathsf{Member}(z \neq \mathsf{self} \to z.p = \mathsf{true})$, which is indeed the weakest precondition we informally expect.

Retraction of beliefs is axiomatized similarly. A command $\mathsf{retract}(e)$ with precondition P and postcondition Q results in the verification condition

$$P \to Q[\mathsf{false}/\mathsf{self}.e]\ .$$

Finally, we consider assignments that involve a query of the belief base. That is, we investigate statements of the form $u := \mathsf{query}(e)$. Note that u is a variable with type bool in this context. Observe that a query does not alter the belief base. Therefore we can test in the state before the assignment by means of the expression $\mathsf{self}.e$ what the value will be that is assigned to u. This observation leads to the following verification condition. If P is the precondition of this statement and Q is its postcondition, then the corresponding verification condition is

$$P \to Q[\mathsf{self}.e/u]\ .$$

The substitution $[\mathsf{self}.x/u]$ completely corresponds to the usual notion of structural substitution. We have $u[\mathsf{self}.e/u] \equiv \mathsf{self}.e$, and $v[\mathsf{self}.e/u] \equiv v$ for every local variable v that syntactically differs from u.

4.3 Reasoning about Requests

In this section, we will explain in detail how reasoning about communication in our agent language takes place. We will develop a variant of the cooperation test described in [13]. The cooperation test describes the verification conditions that result from the annotation of request and answer statements.

Three statements are involved in a communication step. The first statement is a request of an agent of the form $u := e_0.m(e_1, \ldots, e_n)$. The second statement is a corresponding answer statement $\mathsf{answer}(m_1, \ldots, m_n)$ such that $m \in \{m_1, \ldots, m_n\}$. This statement must occur in the internal activity of agent class $[\![e_0]\!]$. The third statement is the body S of the implementation of method m in class $[\![e_0]\!]$. That is the statement that will be executed as a result of the request.

The correspondence between a request and the implementation of the method is statically given. We assume that $m(u_1, \ldots, u_n)\{\ S\ \texttt{return}\ e\ \}$ is the implementation of method m in some agent class A. Therefore it corresponds to the call $u := e_0.m(e_1, \ldots, e_n)$ if $[\![e_0]\!]$ is A.

However, we cannot (in general) statically determine to which answer statement a request corresponds (if any). Therefore we have to consider all possibilities. The cooperation test consists of two verification conditions for every *syntactically matching* request/answer pair. An answer statement $\mathsf{answer}(m_1, \ldots, m_n)$ is said to match syntactically with a request $u := e_0.m(e_1, \ldots, e_n)$ if:

- the answer statement occurs in class $[\![e_0]\!]$;
- $m \in \{m_1, \ldots, m_n\}$.

So let us consider a syntactically matching request/answer pair and a corresponding implementation $m(u_1, \ldots, u_n)\{\ S\ \texttt{return}\ e\ \}$. We assume the following pre- and postconditions.

- $\{P[\bar{v}, \bar{z}]\}\ u := e_0.m(e_1, \ldots, e_n)\ \{Q\}$
- $\{P'[\bar{v}', \bar{z}']\}\ \mathsf{answer}(m_1, \ldots, m_n)\ \{Q'\}$
- $\{P''\}\ S\ \{Q''\}$

The assertions P, P' and Q, Q' may be arbitrary assertions. The specification of the method body S should satisfy certain restrictions. The only local variables that the precondition P'' can refer to are self and the formal parameters of the method. All other local variables are out of scope in P''. The postcondition Q'' is not allowed to mention local variables at all.

The precondition of a request and an answer statement may also specify a list of local variables $\bar{v}$ and a corresponding list of logical variables $\bar{z}$ of equal length. The logical variables $\bar{z}$ are used to temporarily replace the local variables $\bar{v}$ in the context of the implementation. That is needed because the local variables of the request and answer statement are out of scope in the specification of the implementation. We provide an example of the use of this feature below.

Recall that the activity of the requesting agent is suspended during the execution of the request. This means that its local variables and its internal state cannot be altered during the request. The execution of the internal activity

of the answering agent is also suspended while it executes the corresponding method. This implies that the local variables of its internal activity also remain unchanged during the request. However, the execution of the method can result in changes to instance variables of the answering agent.

The state of the program at the start of the request is described by the conjunction of P and P'. Together, they should imply that the precondition of the implementation holds. This is checked by the following verification condition.

$$P \wedge (P'[e_0/\mathsf{self}]) \wedge e_0 \neq \mathsf{self} \rightarrow (P''[e_0, \bar{e}, \mathsf{self}/\mathsf{self}, \bar{u}, \mathsf{caller}][\bar{v}, \bar{v}'/\bar{z}, \bar{z}']) \qquad (1)$$

Note that (1) is the above mentioned implication augmented with several substitutions and the clause $e_0 \neq \mathsf{self}$. This latter clause reflects the fact that an agent cannot communicate with itself. The verification condition (1) is the first half of the cooperation test.

The first substitution $[e_0/\mathsf{self}]$ resolves the possible clash between occurrences of self in both P and P'. These occurrences do not refer to the same object. Therefore we replace the keyword self in P' by the expression e_0 since e_0 is a reference to the answering agent from the perspective of the requesting agent.

Observe that it is also possible that both P and P' mention the same local variable. This should be resolved by replacing the occurrences of this local variable in P' by a fresh logical variable. A similar warning applies to the post-conditions Q and Q'.

The composite substitution $[e_0, \bar{e}, \mathsf{self}/\mathsf{self}, \bar{u}, \mathsf{caller}]$ denotes a simultaneous substitution. First of all, it involves the same substitution of self by e_0 as mentioned above. Secondly, it contains a substitution of the formal parameters $\bar{u} = u_1, \ldots, u_n$ by the actual parameters $\bar{e} = e_1, \ldots, e_n$. Finally, it specifies a substitution of the logical variable caller by self. The logical variable caller is an implicit argument of the call similar to the keyword self. It always receives the value of self from the perspective of the requesting agent. Thus we always have a reference to this agent in the context of the implementation. This reference may only be used in the annotation. We will provide an example of its use at the end of this section.

The substitution $[\bar{v}, \bar{v}'/\bar{z}, \bar{z}']$ denotes the simultaneous replacement of the logical variables in $\bar{z}$ and $\bar{z}'$ by the corresponding local variables in $\bar{v}$ and $\bar{v}'$. It captures the fact that local variables of the request and the answer statement are modelled in the context of the implementation by logical variables.

The state upon termination of the implementation is described by the post-condition Q''. It should imply that both Q and Q' hold. More precisely, they should be related in the following way.

$$(Q''[e_0, \bar{e}, \mathsf{self}/\mathsf{self}, \bar{u}, \mathsf{caller}][\bar{v}, \bar{v}'/\bar{z}, \bar{z}']) \wedge e_0 \neq \mathsf{self} \rightarrow (Q[\mathsf{result}/u]) \wedge (Q'[e_0/\mathsf{self}]) \qquad (2)$$

The verification condition (2) is the final part of the cooperation test.

The substitutions in (2) are analogue versions of those in (1). The only new substitution $[\mathsf{result}/u]$ handles the passing of the result value. We assume that the result value e is implicitly assigned to the special-purpose logical variable

result upon termination of S. The substitution $[\mathsf{result}/u]$ in turn symbolically assigns the value of result to u.

The following example shows many features of the cooperation test. We assume that a particular agent class has the following method implementation.

$$\mathrm{add}(u)\{\ v := \mathsf{query}(u);\ \mathsf{assert}(u);\ \texttt{return}\ v;\ \}$$

The method add first checks if belief u already follows from its belief base. It returns this information to the caller after adding belief u to its belief base. This method has the following precondition (3) and postcondition (4).

$$\mathsf{self}.u = z \wedge \mathsf{self} = A \wedge \mathsf{caller}.x = X \tag{3}$$

$$A.u \wedge \mathsf{result} = z \wedge \mathsf{caller}.x = X \tag{4}$$

Note that the specification also states that the instance variable x of the caller is invariant over the request. Here, x is an arbitrary instance variable of the requesting agent, and X is a logical variable of the same type.

The formulas (3) and (4) correspond to P'' and Q'' above. We assume in this example that the internal activity of the same class contains an answer statement that is annotated as follows.

$$\{\mathsf{true}\}\ \mathsf{answer}(\mathrm{add})\ \{\mathsf{true}\}$$

Thus both P' and Q' in our verification conditions will be true. The rather weak annotation of this command suffices for our purpose. The specification of the corresponding call is more interesting.

$$\{z = a.p \wedge \mathsf{self}.x = X\ [a, A]\}\ b := a.\mathrm{add}(p)\ \{a.p \wedge b = z \wedge \mathsf{self}.x = X\}$$

The precondition $z = a.p \wedge \mathsf{self}.x = X$ corresponds with P above, and $a.p \wedge b = z \wedge \mathsf{self}.x = X$ is the postcondition Q. This specification is a translation of the method specification to the context of the requesting agent. The singleton list a is an instantiation of the sequence $\bar{v}$ in (1) and (2) above. Similarly, A corresponds to the sequence $\bar{z}$. This suffices because a is the only local variable in the specification of the request statement.

We now instantiate the verification conditions in (1) and (2) as given above with the pre- and postconditions of the example statements. This results in the following two verification conditions.

$$(z = a.p \wedge \mathsf{self}.x = X) \wedge (\mathsf{true}[a/\mathsf{self}]) \wedge a \neq \mathsf{self} \rightarrow \\ ((\mathsf{self}.u = z \wedge \mathsf{self} = A \wedge \mathsf{caller}.x = X)[a, p, \mathsf{self}/\mathsf{self}, u, \mathsf{caller}][a/A]) \tag{5}$$

$$((A.u \wedge \mathsf{result} = z \wedge \mathsf{caller}.x = X)[a, p, \mathsf{self}/\mathsf{self}, u, \mathsf{caller}][a/A]) \wedge a \neq \mathsf{self} \rightarrow \\ ((a.p \wedge b = z \wedge \mathsf{self}.x = X)[\mathsf{result}/b]) \wedge (\mathsf{true}[a/\mathsf{self}]) \tag{6}$$

After applying the substitutions we end up with the following implications.

$$(z = a.p \wedge \mathsf{self}.x = X) \wedge \mathsf{true} \wedge a \neq \mathsf{self} \rightarrow (a.p = z \wedge a = a \wedge \mathsf{self}.x = X)$$

$$a.p \wedge \mathsf{result} = z \wedge \mathsf{self}.x = X \wedge a \neq \mathsf{self} \rightarrow (a.p \wedge \mathsf{result} = z \wedge \mathsf{self}.x = X) \wedge \mathsf{true}$$

Both verification conditions are evidently valid.

4.4 Interference Freedom

As explained above, the final component of the system is an interference freedom test. It tests if the validity of the proof outline in a particular agent class is not invalidated by the execution of commands by other agents (possibly of the same class). Such a test is needed because assertions in general describe properties of the global state. What we propose in this section is an adaptation of the interference freedom test in [12].

The interference freedom test ensures that every assignment S executed by an agent does not interfere with an assertion Q in another agent (but not necessarily in another class). In other words, Q should be invariant over execution of S. This test generates a verification condition for every pair of an assertion Q and a statement S that possibly interferes.

Inspection of the commands in our language learns us that there are four sorts of statements that might interfere. First of all, assignments to instance variables have to be considered. Secondly, creation of new agents might invalidate assertions. For example, because an assertion states that P holds for all existing agents of a sort a. But then P should also hold for the new agent u on completion of $u := \mathsf{new}(a)$. Finally, we remark that both assertion and retraction of beliefs possibly influence the validity of assertions. Observe that assignments to local variables cannot interfere with the states of other agents.

The definition of the interference freedom tests for the different sorts of statements is straightforward given the substitutions that we defined in the previous sections. Let the assignment $\mathsf{self}.x := e$ have precondition P, then Q is invariant over $\mathsf{self}.x := e$ if:

$$(Q[z/\mathsf{self}] \wedge P \wedge z \neq \mathsf{self}) \rightarrow Q[z/\mathsf{self}][e/\mathsf{self}.x] \ .$$

To avoid clashes between the keywords self in Q and P we substitute self in Q by a fresh logical variable z. The statement can only interfere if it is executed in parallel by a different agent. Hence the additional assumption $z \neq \mathsf{self}$.

Similarly, we should check that for any assignment $u = \mathsf{new}(a)$ with precondition P we have that Q is invariant. That is, we should check

$$(Q[z/\mathsf{self}] \wedge P \wedge z \neq \mathsf{self}) \rightarrow Q[z/\mathsf{self}][\mathsf{new}(a)/u] \ .$$

The test for a statement $\mathsf{assert}(e)$ (7) and a command $\mathsf{retract}(e)$ (8) are also similar.

$$(Q[z/\mathsf{self}] \wedge P \wedge z \neq \mathsf{self}) \rightarrow Q[z/\mathsf{self}][\mathsf{true}/\mathsf{self}.e] \tag{7}$$

$$(Q[z/\mathsf{self}] \wedge P \wedge z \neq \mathsf{self}) \rightarrow Q[z/\mathsf{self}][\mathsf{false}/\mathsf{self}.e] \tag{8}$$

The inference freedom test completes our proof method for the presented object-oriented agent language.

5 Conclusions

Reasoning about agent communication is a relatively new topic. In [14] the verification problem of agent communication is defined and examined at a rather

high level of abstraction. The different components of a verification framework for agent communication are outlined in [15]. An actual verification method for agent communication is developed in [16] that is based on the synchronous handshaking mechanism of Hoare's paradigm of CSP (Communicating Sequential Process) and the information-processing aspects of CCP (Concurrent Constraint Programming). In this framework, agent communication proceeds through passing information along bilateral channels rather than by requests like in the current paper.

In this paper we have described (to the best of our knowledge) a first proof method based on assertions for an object-oriented coordination language for multi-agent systems. To our opinion this work presents a promising first step towards a further integration of OO and agent technology (for example, 'agentifying' inheritance and subtyping) and the application of formal methods to complex agent systems. Additionally, this work also provides a promising basis for the application of concepts like compositionality developed in the semantics and proof theory of concurrent systems [17].

One of the main proof theoretical challenges for future developments concerns an integration of a new class of modal logics [18] for representing the beliefs of agents which capture the object-oriented mechanism of dynamic referencing of agents by means of corresponding parameterized modal operators. A perhaps even more ambitious future goal is to extend the proof method to agent-oriented programming languages like 3APL [19] which involve very expressive mechanisms for belief and goal revision.

Currently, we are working on soundness and completeness proofs. The soundness proof consists of a fairly straightforward induction on the length of the computations. The completeness proof consists of showing that the verification conditions hold for proof outlines which are defined in terms of so-called semantic reachability predicates (see [17]).

In the near future we are also planning to develop tools for the automatic generation of the verification conditions and the use of theorem provers along the lines of [10].

References

1. Rash, J., Rouff, C., Truszkowski, W., Gordon, D., Hinchey, M., eds.: Formal Approaches to Agent-Based Systems (Proc. FAABS 2001). Volume 1871 of LNAI. Springer (2001)
2. Hinchey, M., Rash, J., Truszkowski, W., Rouff, C., Gordon-Spears, D., eds.: Formal Approaches to Agent-Based Systems (Proc. FAABS 2002). Volume 2699 of LNAI. Springer (2003)
3. Wooldridge, M., Ciancarini, P.: Agent-oriented software engineering: The state of the art. In Wooldridge, M., Ciancarini, P., eds.: Agent-Oriented Software Engineering. Volume 1957 of LNCS. (2001) 1–28
4. Meyer, J.J.C.: Tools and education towards formal methods practice. In Hinchey, M., Rash, J., Truszkowski, W., Rouff, C., Gordon-Spears, D., eds.: Formal Approaches to Agent-Based Systems (Proc. FAABS 2002). Volume 2699 of LNAI. (2003) 274–279

5. van Eijk, R., de Boer, F., van der Hoek, W., Meyer, J.J.C.: Generalised object-oriented concepts for inter-agent communication. In Castelfranchi, C., Lesprance, Y., eds.: Intelligent Agents VII. Volume 1986 of LNAI. (2001) 260–274
6. Hoare, T.: Assertions. In Broy, M., Pizka, M., eds.: Models, Algebras and Logic of Engineering Software. Volume 191 of NATO Science Series. IOS Press (2003) 291–316
7. Floyd, R.W.: Assigning meaning to programs. In: Proc. Symposium on Applied Mathematics. Volume 19., American Mathematical Society (1967) 19–32
8. Meyer, B.: Eiffel: The Language. Prentice-Hall (1992)
9. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12** (1969) 576–580
10. de Boer, F., Pierik, C.: Computer-aided specification and verification of annotated object-oriented programs. In Jacobs, B., Rensink, A., eds.: FMOODS V, Kluwer Academic Publishers (2002) 163–177
11. Pierik, C., de Boer, F.S.: A syntax-directed Hoare logic for object-oriented programming concepts. In Najm, E., Nestmann, U., Stevens, P., eds.: Formal Methods for Open Object-Based Distributed Systems (FMOODS) VI. Volume 2884 of LNCS. (2003) 64–78
12. Owicki, S., Gries, D.: An axiomatic proof technique for parallel programs i. Acta Informatica **6** (1976) 319–340
13. Apt, K., Francez, N., de Roever, W.: A proof system for communicating sequential processes. ACM Transactions of Programming Languages and Systems **2** (1980) 359–385
14. Wooldridge, M.: Semantic issues in the verification of agent communication. Autonomous Agents and Multi-Agent Systems **3** (2000) 9–31
15. Guerin, F., Pitt, J.: Verification and compliance testing. In Huget, M.P., ed.: Communication in Multiagent systems: Agent communication languages and conversation policies. Volume 2650 of LNAI. (2003) 98–112
16. Eijk, R.v., Boer, F.d., Hoek, W.v.d., Meyer, J.J.: A verification framework for agent communication. Journal of Autonomous Agents and Multi-Agent Systems **6** (2003) 185–219
17. de Roever, W.P., de Boer, F., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J.: Concurrency Verification. Volume 54 of Cambridge Tracts in Theoretical Computer Science. Cambridge University Press (2001)
18. Meyer, J.J.C., van der Hoek, W.: Epistemic Logic for AI and Computer Science. Volume 41 of Cambridge Tracts in Theoretical Computer Science. Cambridge University Press (1995)
19. Hindriks, K., de Boer, F., van der Hoek, W., Meyer, J.J.: Agent programming in 3APL. Autonomous Agents and Multi-Agent Systems **2** (1999) 357–401

On Feature Orientation and on Requirements Encapsulation Using Families of Requirements

Jan Bredereke

Universität Bremen, FB 3, P.O. box 330 440, D-28334 Bremen, Germany
brederek@tzi.de
www.tzi.de/~brederek

Abstract. Naive feature orientation runs into problems with large software systems, such as telephone switching systems. With naive feature orientation, a feature extends a base system by an *arbitrary* increment of functionality. Information hiding helps to structure a large software system design into modules such that it can be maintained. We focus on the requirements of a software system. Requirements can be structured analogously to design modules. Naive feature orientation can violate requirements encapsulation. We survey approaches with improved encapsulation, and we show how and when families of requirements can help.

1 Introduction

A feature oriented description of a software system separates a base system from a set of optional features. Each feature extends the base system by an increment of functionality. Feature orientation emphasizes the individual features and makes them explicit. The description of one feature does not consider other extensions of the base system. Any interactions between features are described implicitly by the feature composition operator used.

Feature orientation is attractive. It meets the needs of marketing. Marketing must advertise what distinguishes a new version from its predecessors. Marketing must offer different functionality to different customers, in particular at different prices. Successful marketing also demands a short time to market. This requires that the system can be changed easily. It can be achieved by just adding a new feature. The large body of existing descriptions never needs to be changed.

But naive feature orientation runs into problems with large software systems, such as telephone switching systems. In this paper, we show where naive feature orientation can violate information hiding, and how and when *families of requirements* can help.

2 Feature Orientation

2.1 Naive Feature Orientation

With *naive* feature orientation, a feature extends a base system by an *arbitrary* increment of functionality. The increment is typically chosen to satisfy some new

M.D. Ryan et al. (Eds.): Objects, Agents, and Features, LNCS 2975, pp. 26–44, 2004.

user needs. This selection of user needs happens from a marketing perspective. In particular, the selection is neither particularly aligned to the internal structure of the software system nor to the organization of the system's documented requirements.

Many feature addition operators have been used in practice or proposed on theoretical grounds [1–6]. They typically share the property that they add code in different places of the base system as needed. They are therefore operators of a syntactic nature.

A canonical example is the structure of the Intelligent Network (IN) [7–9]. The IN is the telephone switching industry's currently implemented response to the demand for new features. This example demonstrates the naive feature orientation nicely. This remains true even if the IN might be replaced by emerging architectures eventually, such as Voice over IP (VoIP).

The IN specifies the existence of a Basic Call Process (BCP) and defines sets of features. Examples of IN features are listed in Fig. 1. When a feature is triggered, processing of the BCP is suspended at a Point of Initiation (POI), see Fig. 2. The feature consists of Service-Independent Building Blocks (SIBs), chained together by Global Service Logic. Processing returns to the BCP at a Point of Return (POR). The Basic Call Process consists of two automata-like descriptions, one for the originating side of a call, and one for the terminating side of a call, see Fig. 3. In these, a feature can be triggered at a so-called Detection Point, and processing can resume at more or less any other Detection Point. This allows a feature to modify basic call processing arbitrarily.

There has been considerable research effort on feature composition operators. In particular, in the FIREworks project [10, 11] (Feature Interactions in Requirements Engineering), various feature operators were proposed and investigated. These operators successfully reflect the practice of arbitrary changes to the base system. The theoretical background is the superimposition idea by Katz [12]: one specifies a base system and textual increments, which are composed by a precisely defined (syntactic) composition operator.

A feature is inherently *non-monotonous* [15]. Most features really change the behaviour of the base system. That is, a feature not only adds to the behaviour of the base system, or only restricts the behaviour of the base system. For example, in telephony a call forwarding feature both restricts and adds to the behaviour. It prevents calls to the original destination, and it newly makes calls to the forwarded-to destination. Therefore, a refinement relation is not suitable to describe adding a feature.

2.2 Feature Interaction Problems in Telephone Switching

It turns out that severe feature interaction problems appear if one applies a naive feature oriented approach to a large software system, such as a telephone switching system. It is relatively easy to create a new feature on its own and make it work. But it becomes extremely difficult to make all the potential combinations of the optional features work as the users and providers expect. The telecom industry complains that features often interact in an undesired way [1–

Abbreviated dialling	Customized ringing
Attendant	Destinating user prompter
Authentication	Follow-me diversion
Authorization code	Mass calling
Automatic call back	Meet-me conference
Call distribution	Multi-way calling
Call forwarding	Off net access
Call forwarding on busy/don't answer	Off net calling
Call gapping	One number
Call hold with announcement	Origin dependent routing
Call limiter	Originating call screening
Call logging	Originating user prompter
Call queueing	Personal numbering
Call transfer	Premium charging
Call waiting	Private numbering plan
Closed user group	Reverse charging
Consultation calling	Split charging
Customer profile management	Terminating call screening
Customized recorded announcement	Time dependent routing

Fig. 1. The features in the Intelligent Network (version CS 1).

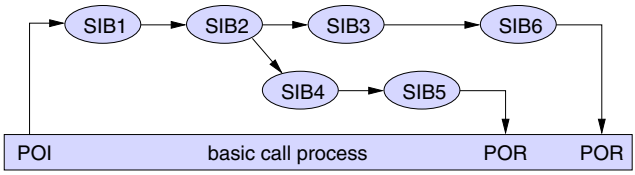


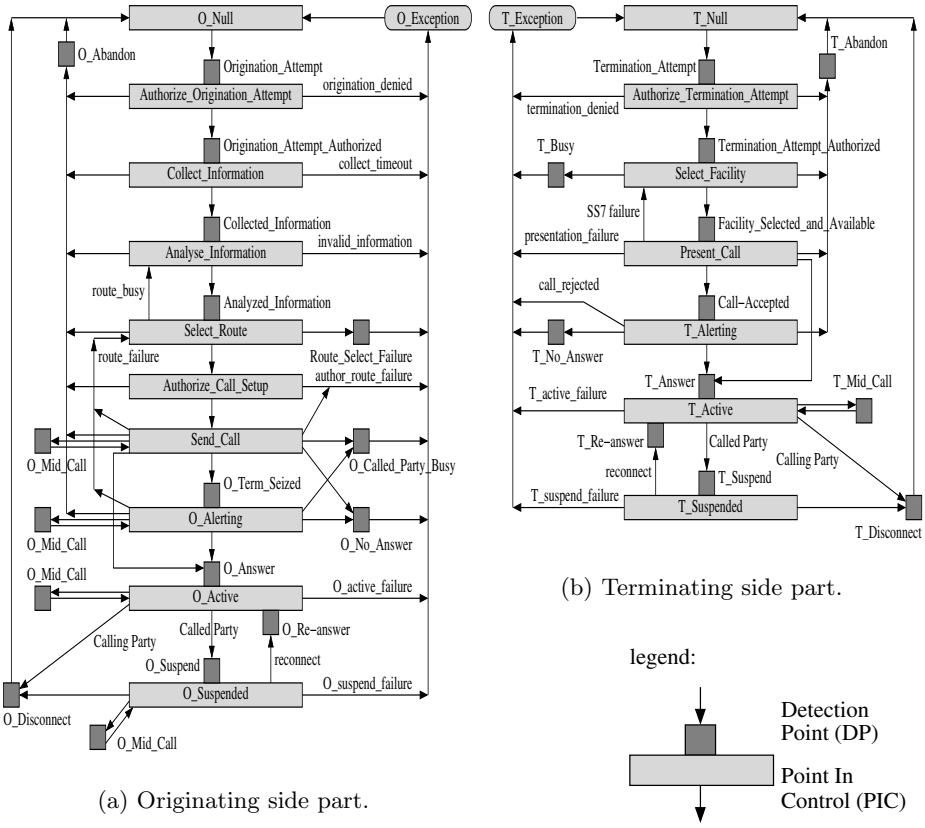
Fig. 2. Feature-oriented extension in the Intelligent Network (from [13, p. 3]).

6]. There are already hundreds of telephony features. The combinations cannot be checked anymore because of their sheer number. Undesired interactions annoy the telephone users, and the users are not willing to accept many of them. The users expect reliability from a telephone system much more than from other software-intensive systems such as desktop PCs.

One typical example of a telephony feature interaction occurs between a Calling Card feature and a Voice Mail feature.

We had once a calling card from Bell Canada. It allowed us to make a call from any phone and have the call billed to the account of our home's phone. We had to enter an authentication code before the destination number to protect us against abuse in case of theft. For ease of use, we could make a second call immediately after the first one without any authorization, if we pressed the "#" button instead of hanging up.

We also had a voice mail service from Meridian at work. A caller could leave a voice message when we couldn't answer the phone. We could check for messages later, even remotely. For a remote check, we had to call an access number, dial



(a) Originating side part.

(b) Terminating side part.

Fig. 3. The Basic Call State Model of the Intelligent Network (version CS 2, after [14]).

our mailbox number and then a passcode. At the end of both the mailbox number and the passcode, we had to press the "#" button.

The interpretations of the "#" button were in conflict between these two features. The calling card feature demanded that the call should be terminated. The voice mail feature demanded that the call should be continued, and that the authorization went on with the next step. This particular feature interaction was resolved by Bell. The calling card feature required that the "#" button was pressed at least two seconds to terminate the call.

A *feature interaction* occurs when the behaviour of one feature is changed by another feature. This is a commonly accepted informal definition.

Not all feature interactions are undesired. Some features have increased value together with other features. For example, a short code to re-dial the last number dialled saves typing. This is even more helpful when one uses a (long) dialling prefix that selects an alternative, cheaper long-distance carrier. Some features are even intended to improve a system that has specific other features. (Of

course, this violates the "pure" feature oriented approach.) For example, a calling number delivery blocking feature interacts with a calling number delivery feature. The latter displays the caller's number at the callee's phone. The former prevents a caller's number to be displayed anywhere for privacy reasons.

Cameron *et al.* [16] have categorized the causes of feature interaction problems in their seminal benchmark paper: violation of feature assumptions, limitations on network support, and intrinsic problems in distributed systems. Some violated feature assumptions are on naming, data availability, the administrative domain, call control, and the signalling protocol. Limitations on network support occur because of limited customer premises equipment signalling capabilities and because of limited functionalities for communications among network components. Some intrinsic problems in distributed systems are resource contention, personalized instantiation, timing and race conditions, distributed support of features, and non-atomic operations.

A rather comprehensive survey of approaches for tackling feature interaction problems was done recently by Calder *et al.* [17]. Despite some encouraging advances, important problems still remain unsolved. The rapid change of the telecommunications world even brings many new challenges.

A new view on the causes of feature interaction problems was a main result of the recent seventh Feature Interaction Workshop [1]: in order to resolve a conflict at a technical level, we often need to look at the social relations between users to either disambiguate the situation or mediate the conflict. Zave [18] pointed out that features should be purposes, not mechanisms. Gray *et al.* [19] found that busy is a person's state, not a device's state – and the answer depends on who is asking. We [20] showed that many feature interaction problems arise because the users fail to abstract the system to the relevant aspects correctly. Other authors went into similar directions. In a panel discussion, Logrippo compared feature interaction resolution to legal issues.

3 Information Hiding Definitions

Information hiding helps to structure a large software system design into modules such that it can be maintained. We now introduce some definitions from the literature as a base for our further discussion.

A *module* in the information hiding sense [21–23] is a work assignment to a developer or a team of developers. (There are *many* other meanings of this word, we use this meaning only here.) Such a work assignment should be as self-contained as possible. This reduces the effort to develop the system, it reduces the effort to make changes to the system later, and it improves comprehensibility. A successful software system will be changed many times over its life time. When some design decision must be changed, a change should be necessary in one module only. A design decision usually must be changed when some requirement changes.

The *secret* of a module is a piece of information that might change. No other module may rely on the knowledge of such a secret. Sometimes we distinguish

between a primary and a secondary secret. A primary secret is hidden information that was specified to the software designer. A secondary secret is a design decision made by the designer when implementing the module that hides the primary secret.

The *interface* between modules is the set of *assumptions* that they make about each other. This not only includes syntactic conventions, but also any assumptions on the behaviour of the other modules. A developer needs to know the interface of a module only in order to use its services in another module.

There can be a *hierarchy of modules*. We need it for large systems. Its structure is documented in a *module guide*. The module guide describes the module structure by characterizing each module's secrets.

A fundamental *criterion for designing the module structure* of a software system is: identify the requirements and the design decisions that are likely to change, and encapsulate each as the secret of a separate module. If such a module is too large for one developer, the approach must be applied recursively. This leads to making the most stable design decisions first and those most likely to change last. The three top-level modules for almost any software system should be the hardware/platform-hiding module, the behaviour-hiding module and the software decision module. These modules must then be decomposed recursively, depending on the individual system. The structure presented in [23] might serve as a template.

An *abstraction* of a set of entities is a description that applies equally well to any one of them. An *abstract interface* is an abstraction that represents more than one interface; it exactly and only consists of the assumptions that are included in all of the interfaces that it represents. A *device interface module* is a set of programs that translate between the abstract interface and the actual hardware interface [24]. Having an abstract interface for a device allows to replace the device during maintenance by another, similar model with a different hardware interface, without changing more than one module.

Object orientation allows to use information hiding by realizing modules through the mechanism of the class.

Information hiding enables to design software for ease of *extension and contraction*. Design for change must include the identification of the minimal subset that might conceivably perform a useful service, and it must include the search for a set of minimal increments to the system [25]. The emphasis on minimality stems from the desire to avoid components that perform more than one function.

The *relation "uses"* among programs (i.e., pieces of code) describes a correctness dependency. A program A uses B if correct execution of B may be necessary for A to complete the task described in A's specification. We can facilitate the extension and contraction of a software, if we design the uses relation to be a hierarchy (i.e., loop-free), and if we restrict it as follows. A is allowed to use B only when all of the following conditions hold: (1) A is essentially simpler because it uses B. (2) B is not substantially more complex because it is not allowed to use A. (3) There is a useful subset containing B and not A. (4) There is no conceivably useful subset containing A but not B.

Information hiding is also a base for the design and development of *program families*. A set of programs constitutes a family, whenever it is worthwhile to study programs from this set by first studying the common properties of the set and then determining the special properties of the individual family member [26]. One worked-out approach is [27].

4 Requirements

We focus on the *requirements* of a software system for the discussion of feature orientation. Feature interactions often already arise in the requirements documents. When these documents are a *complete* description of the behaviours and of other interesting properties, then *all* feature interaction problems are present in the requirements documents at least inherently. Therefore, they should be tackled already there.

4.1 Families of Requirements

A *family of requirements* is a set of requirements specifications for which it pays off to study the common requirements first and then the requirements present in few or individual systems only. In particular, we are interested in families of requirements where only a subset of the family is specified explicitly in the beginning, and where more members are specified explicitly incrementally over time.

A family of requirements means that we have several versions of requirements. Requirements can and should be put under configuration management analogously to software. The "atomic objects" are properties.

The right size of the properties, when taken as the atomic objects of configuration management, depends on the size of the family. We must split up the requirements specification into small properties when a family of requirements has a large number of potentially specified members. We want to avoid to specify the same aspect A in two different properties. This can happen if we specify two aspects A, B in one property P_1 first and then need to specify A in another property P_2 again, because there is a family member that has A but not B.

In case of doubt, we should make a property in the requirements as small as possible while being useful. This is a safe strategy when we cannot overlook the entire set of family members easily. Such a specified property will be much smaller than the user of a new system or of a new feature usually thinks.

By small, we mean abstract in the above sense. A small property is part of the requirements of as many useful potential systems as possible. The goal is that each time a new member is specified, we will never need to copy and modify any existing property. The new member will only exchange one or more entire properties by one or more other properties.

When we have a large number of requirements and therefore of requirements modules, we need some additional structure. It shall help the reader of a requirements document to find easily the module he/she is interested in. The above

kind of modules is not directly suitable. The above modules are a product of the software *design*. Their secret can be a requirement or a design decision. Their structure is a software design structure. Such artefacts of the software design do not belong into the software requirements. But we can adapt the idea.

A *requirements module* is a set of properties that are likely to change together. A requirements module may be partitioned recursively into sub-modules. Each sub-module then is a set of properties that are even more likely to change together. By "likely to change together", we mean that for many potentially specified members of the family, either all the properties from the set are included, or none. The likeliness increases with the number of family members where this is true.

We propose to *organize requirements by requirements modules*. Those requirements should be grouped together that are likely to change together. It then becomes easier to specify another member of the family explicitly. It is likely that we can take an already specified member, remove one requirements module, and add another requirements module. The latter possibly also has been specified explicitly already for another member.

A criterion for the quality of the organization of the specified requirements modules is how many modules must be changed for obtaining another family member, on the average. These change costs must be weighted with the probability that the change actually occurs.

The above higher-level modules need to be decoupled. For example, there might be family members that interface to a device of kind A, and other family members that interface to a device of kind B. The behaviour of the system is similar for both sets of family members, except for the details of the device.

Our solution is similar to the idea of abstract interfaces in design [24]. We define abstract interfaces in abstraction modules. The advantage of having a device interface requirements module that declares abstract variables and/or events is that the properties of the behaviour modules need to depend only on the stable properties in this module. For example, in telephony it is preferable to base the behaviour on the abstract term of a connection request than on a hook switch being closed by lifting a handset.

We want to have consistent configurations of the requirements document only. When we construct a new configuration, we usually start with an existing, consistent configuration and add and/or remove some properties. One of the difficulties then is to take care of all dependencies among the properties. Further additions and removals may be necessary. Maybe we even need to specify some more properties explicitly. Only then we will arrive at another consistent configuration of properties.

Localizing the changes into one or a few requirements modules helps a lot, but it is not yet sufficient. A property can sometimes depend on other properties from other modules. For example, if entire modules are added or removed, these dependencies must be checked.

A property P_2 *depends* on a property P_1, if P_2 is not well-formed without the presence of P_1. In particular, declarations cause dependencies. For example, P_1

introduces a variable monitored by the system, such as the position of a button. P_2 determines the system behaviour depending on the value of this variable. P_2 would not make sense without the variable being declared.

The dependency relation must be *explicit*. Otherwise, any maintainer not knowing the entire specification by heart must check all requirements for consequences. This is not feasible for large specifications. Therefore, the dependencies should be documented when they are created.

One goal of the explicit dependency hierarchy is that the specifier tries to have as little dependencies as possible. For each dependency, the specifier should check whether it is necessary.

Each property must be formulated such that is a minimal useful increment, as discussed in in Sect. 4.1 above. The dependencies are, in our experience, a good means to check this. A property that depends on many other properties is probably not minimal and could be split up.

The requirements module hierarchy is quite different from the requirements dependency hierarchy. We must take great care to not confuse them. A particular mistake we must avoid is to force the requirements module hierarchy to be the same as the dependency relation. In general, there is not necessarily a correlation between two abstract requirements depending on each other, and being likely to change together. The relationship of requirements modules and requirements dependencies is similar to the relationship of design modules and the design "uses" relation among programs. (See Sect. 3 above.)

Another mistake to avoid is to define the dependency relation between (sub-)modules. The dependency relation is among properties, not among modules. Defining the dependency relation between modules instead of between properties would introduce additional, artificial dependencies.

4.2 Requirements Modules and Features

A *feature* is some increment relative to some baseline, and most features are non-monotonous (see Sect. 2.1). Therefore, a feature consists of a set of added properties and of a set of removed properties. In the language of configuration management [28], a feature is a "change", also called a directed delta. In our particular setting with a set of optional (or mandatory) properties, a feature consists of the set of names of properties that must be included and of the set of names of properties that must be excluded. We may say that a feature is a configuration rule.

A feature is not a requirements module. Many approaches use features as requirements modules. But this creates maintenance problems. Features and requirements modules are similar. Both concepts serve to group properties. But there are two marked differences between features and requirements modules:

1. A requirements module is a set of properties (i.e., *one* set), while a feature consists of both added and removed properties.
2. The properties of a module are selected because of their likeliness to change together, *averaged* over the entire family, while the properties of a feature are selected to fit the marketing needs of a *single situation*.

Forcing requirements modules and features to be the same is not advisable. A feature fits the marketing needs of one occasion only, even though perfectly. It is likely to not fit well for the remaining family members. A requirements module supports the construction of all family members well, even though it does not satisfy all the marketing needs of a particular occasion by itself. A few other requirements modules will be concerned, too. In contrast, adding one more feature on top of a large naively feature-oriented system will concern many other features.

A requirements module provides an abstraction, while a feature is a configuration rule for such abstractions.

An example from telephony is the following:

- The 800 feature allows a company to advertise a single telephone number, e.g., 1-800-123-4567. Dialling this number will connect a customer with the nearest branch, free of charge. This feature should be composed of properties from these three requirements modules: a module that provides addresses for user roles, a module that translates a role address to a device address based on the caller's address, and, entirely independently, a module that charges the callee. The feature removes the property that the caller is charged.
- The emergency call feature allows a person in distress to call a well-known number (911 in the U.S., 110 in Germany and in some other European countries, ...) and be connected with the nearest emergency center. This feature will include the properties from the three requirements modules above, and of a few more. For example, there will be properties from a module that allows the callee to identify the physical line the call comes from.
- The follow-me call forwarding feature allows a person to register with any phone line and receive all calls to his/her personal number there. This feature includes properties from the above module which provides addresses for user roles. The other modules are not needed. Instead, we need properties from a module that translates a role address according to a dynamic user preference. We also need properties from a further module to set user preferences dynamically.

Successful marketing needs features such as the above ones. A "user role address" feature would probably sell much worse than the ubiquitous 800 feature. But the above reuse of requirements modules would not be possible in a naive feature-oriented approach.

Two features might easily be incompatible, i.e., the features interact adversely, because one feature includes a certain property while the other feature excludes it. The features, seen as configuration rules, contradict each other.

A solution is to have different configuration priorities for the properties of a feature. We distinguish (at least) the *essential properties* and the *changeable properties* of a feature. An essential property is necessary to meet the expectations evoked by the feature's name. A changeable property is provided only in order to make the requirements specification complete and predictable for the user. For example, a call forwarding feature can be recognized no matter what

the requirements say on whether a the forwarding target can be set by pressing a button sequence or by, e.g., speech recognition. We therefore propose that the specifier of a feature documents explicitly which properties are essential and which are changeable.

5 Evaluation of Other Work with Respect to Requirements Encapsulation

5.1 Naive Feature Orientation and Requirements Encapsulation Violations

Naive feature orientation supports families of requirements, but does not organize the requirements into requirements modules as discussed above. This leads to feature interaction problems. We now show in our canonical example, the Intelligent Network, where the above encapsulation guidelines for the requirements are violated.

The specification of the Intelligent Network is oriented along execution steps. It is hard to specify a property of the IN without saying a lot about the exact sequencing of steps. The Basic Call Process consists of explicit automata with explicit triggering points, and the Service Independent Building Blocks of a feature are chained together by the explicit sequencing of the Global Service Logic. This violates the principle of making any single requirement as small and abstract as possible, and composing the base system and the features from these atomic properties.

The Service Independent Building Blocks (SIBs) provided in the standard [29] are designed to be general in the sense that they offer a lot of functionality. For example, the Charge SIB performs a special charging treatment for a call, and the Algorithm SIB applies a mathematical algorithm to data to produce a data result. Any details of the operations are controlled by run-time parameters. Any concrete system requirements document must specify which charging or calculating operations these SIBs support, respectively. From then on, it is likely that there will come up another operation not yet supported. This will require a change of the SIB concerned. This in turn threatens to break all other features using this SIB. SIBs therefore are usually not a unit of most abstract requirement.

The Basic Call Process itself violates the principle of making any single requirement as small and abstract as possible. It specifies many different aspects at the same time, as could be seen above. Instead of allowing for small requirements to be taken out and in, a monolithic specification provides hooks for changes of its behaviour. Few properties of its behaviour will be valid for all sets of features. It is hard to design a feature on top of this monolithic base system that will not break for some combination of features. If the base system would consist of smaller, explicitly stated properties with explicitly stated dependencies, then it would be easier to see which features are affected when a new feature removes a certain property.

One example is the step from the two-party call to the n-party session. The Basic Call Process is written in terms of the two-party call. Nevertheless, the Intelligent Network allows to combine several call legs. The n-party call is necessary for such features as Consultation Call, Conference Call, and Call Forwarding. Many features and SIBs are designed with the two-party call in mind, though. For example, the Screen SIB compares a data value against a list. If it is used to specify originating call screening, the screening can fail. Call Forwarding can translate the dialled number several times before making a connection. A single instance of the Screen SIB will check only one of the numbers. Even though the Basic Call Process insinuates that there is exactly one terminating side (Fig. 3), this property is not true for all systems.

The user interface is likely to change, nevertheless its concerns are spread out. This is so despite there being a User Interaction SIB that is intended to perform the user interaction for one feature. Most of the IN features need to interact with a user. This interaction must be possible through a scarce physical interface: twelve buttons, a hook switch, and a few signal tones. Ten of the buttons are used already by the base system. Physical signals must therefore be reused in different modes of operation. But the definitions of several features implicitly assume exclusive access to the user's terminal device. There is no single requirement that specifies the scheme how multiple features coordinate the access. The above interaction between a calling card feature and a voice mail feature is a consequence. Both features assume exclusive access to the "#" button. Details of the user interface are specified at the bottom of the requirements, even though they are likely to change. We discuss this in more detail in [30, 31].

5.2 Approaches with Improved Requirements Encapsulation

There are many approaches that encapsulate requirements better. We now sketch some of them in the light of requirements encapsulation. Even if some of these approaches use the word "feature", mostly they mean a module that encapsulates a secret. However, none distinguishes features and modules explicitly.

The CoRE (Consortium Requirements Engineering) method [32–34] allows to specify requirements for avionic and other safety-critical systems. A major goal is to plan for change by using information hiding for the requirements. CoRE is based on the functional documentation (four-variable model) approach [35]. It adds additional structure to the requirements document by grouping variables, modes and terms into classes. This borrows from object-orientation. A class has an interface section and an encapsulated section. Entities not needed by other classes are hidden syntactically inside the encapsulated section. The application of CoRE to a Flight Guidance System rendered valuable experience. The authors found that the requirements for the user interface should have been separated from the requirements for the essential nature of the system, since the user interface is more likely to change. Furthermore, they found in particular that planning for change in a single product is not the same as planning for change in a product family [32]. The requirements should have been organized entirely different for the latter.

The Tina initiative (Telecommunication Information Network Architecture) [36, 37], the Race project (Research and technology development in Advanced Communications technologies in Europe), and the Acts project (Advanced Communications Technologies & Services) developed and improved a new service architecture for telecommunications. These projects have added most of the interesting new abstractions explicitly to the resulting architecture. For example, the explicit distinction between a user and a terminal device splits up the host of properties that can be associated with a directory number in the Intelligent Network. Therefore, the requirements of the base system are more structured than for the Intelligent Network. The drawback is that the architecture is quite far away from the structure of current systems, and a transition would be expensive [30].

The DFC (Distributed Feature Composition) virtual architecture is proposed by Jackson and Zave [38]. It is implemented in an experimental IP telecommunication platform called BoxOS [39]. It allows to compose features in a pipe-and-filter network. The filter boxes are relatively simple. This is in accordance with the principle of small requirements. Also, several new abstractions are explicitly supported, for example multi-party sessions and the distinction among users, the different roles they play, and the different terminal devices they may use. A strong point of BoxOS is that it can inter-operate with the existing telephone network. However, part of its functionality is lost for these calls, naturally.

An Agent Architecture is outlined by Zibman et. al. [40]. It separates several concerns explicitly. There are four distinct types of agents: user agents, connection agents, resource agents, and service agents. This separates user and terminal concerns. The terminal resource agent encapsulates the user interface details, such as the signal syntax. The distinct user and connection agents separate call and connection concerns. The user agents bring the session abstraction with them. The connection agents coordinate multiple resource agents. The resource agent separates resource management from both session control and from the services. It was a design goal that the introduction of new services should not require modifications of existing software. Therefore POTS is represented by a single service agent even though POTS really comprises several distinct concerns.

Aphrodite is an agent-based architecture for Private Branch Exchanges that has been implemented recently [41]. Each entity, device and application service is represented as an agent. Agents are therefore abstractions. The often-changing details of the behaviour of an agent are specified as policies. Policies can be changed easily since they are stored as data in a table. It is an explicit goal to make features small. For example, “transfer” is no longer a feature, but made up of three different smaller features: “invoke transfer”, “try transfer”, and “offer transfer”. Another stated goal is to make the assumptions explicit that features make. Also, many new abstractions are already incorporated in the base system as “internal features”.

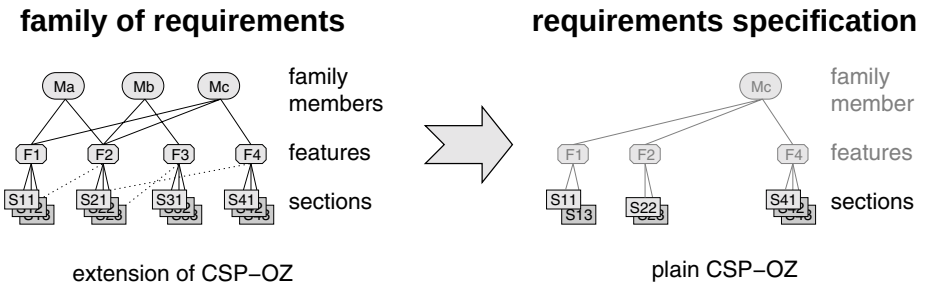


Fig. 4. Generating family members from a family document.

6 An Approach with Families of Rigorous Software Requirements

We have investigated how explicit families of software requirements can facilitate the maintenance task. We showed how the user interface can be encapsulated in a requirements specification of a family of telephone systems [31]. We applied the above requirements encapsulation guidelines in a case study on telephone switching requirements [42, 43].

In [31], we showed how the user interface can be encapsulated in a requirements specification of a family of telephone systems. We proposed to distinguish a syntactic and a semantic level of user interaction. The behavioural requirements should be specified at the semantic level only. Semantic signals should reflect a user's decision to perform some action, or a user's perception that some other user or the system has decided to perform some action. Examples for semantic signals could be "VoiceMailLogin" (for voice mail) and "ReleaseAndReconnect" (for credit card calling). These semantic signals must eventually be mapped to syntactic signals like "flash hook", "#", signal tones, and so on. The mapping should be encapsulated into only one design module, the user interface module. We sketched how such a user interface module could be integrated into the current Intelligent Network architecture.

In [42, 43], we applied the above requirements encapsulation guidelines in a case study on telephone switching requirements. We used a constraint-oriented specification style. All constraints are composed by logical conjunction. We made each constraint as small as usefully possible.

We specified the requirements in the formalism CSP-OZ [44, 45], which we extended by means to specify a family of requirements. All family members are specified in one document. A family member is composed of a list of features. A feature consists of a set of modules and of a list of modules "to remove". A module is represented in CSP-OZ by the formal construct of a section. Each module, i.e., section, holds one abstract requirement. Figure 4 gives an overview. The formalism forces the specifier of any section to state on which other sections it depends. There can be a dependency because the section uses a definition from the other section.

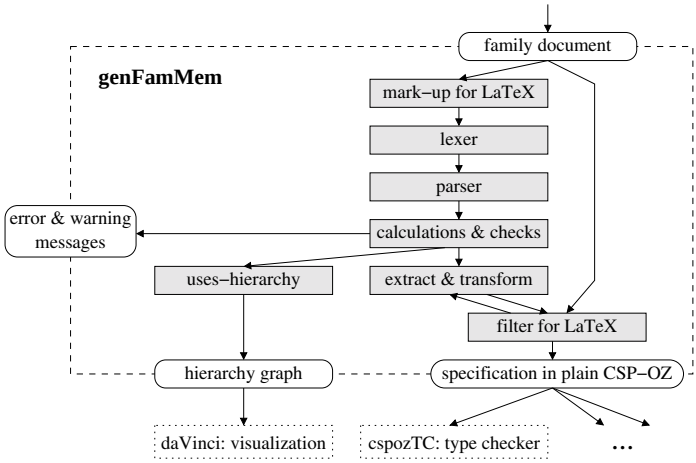


Fig. 5. Data flow structure of the genFamMem tool.

Our formalism also forces the original specifier of a feature to state whether a property is essential for the feature or not. A feature's changeable properties can be removed from the system by another feature through a suitable operator. This allows for non-monotonous changes. But only entire properties, i.e., sections, can be removed and added.

There is a formal semantics both for CSP-OZ [44] and for our additional family construct [46].

We implemented a supporting tool [42, 46]. The tool generates individual family members from the family document as needed, it extracts and displays the dependencies among sections and among features, and it performs type checks on the family constructs. Figure 5 shows its data flow structure. Our formalism imposes some type rules. For example, a section must not be removed if another section from another feature depends on it. Some kinds of feature interactions therefore become type errors. The tool also checks further, heuristic rules that indicate probable feature interaction problems.

We specified the requirements for a telephone switching system in a case study [42, 43]. The case study currently comprises about 40 pages of commented formal specification, with about 50 sections in nine features, including the base system. The communication between the users and the system was specified in terms of semantic signals entirely, as discussed above.

The specification introduces the three notions of "telephone device", "human", and "user role" explicitly and early. As a consequence, the well-known feature interaction problems between call screening and call forwarding vanish. Call screening now appears as two different features: device screening and user screening. Similarly, call forwarding is differentiated into a re-routing when a human moves to another device, and into the transfer of a user role to another human. All combinations of screening and forwarding now work without adverse interactions.

Nevertheless, our tool found a problem between the two screening features. Its heuristic check issued a warning that both features remove the same section. And indeed, a manual inspection showed that the new constraints introduced by the two features (as a replacement for the removed section) contradicted each other. We then could resolve the issue by specifying explicitly the joined behaviour at this point.

7 Discussion

Our focus on requirements roots in the basic engineering principle of design by documentation. Engineers draw blueprints before construction, and they keep them up-to-date. Accordingly, we document requirements explicitly, including the information necessary for changing them.

The organization of the requirements affects the ease of their maintenance. Feature orientation meets the needs of marketing. But naive feature orientation does not scale. We transferred the information hiding principle from design to requirements. Modules of abstract requirements are a base for families of requirements. Families of requirements are our approach to feature orientation.

We found that a feature is not the same as a requirements module. A requirements module provides an abstraction, while a feature is a configuration rule for such abstractions, chosen for marketing. Without this distinction, it becomes harder to express abstractions with long-term value.

A *policy* is very similar to a feature in the sense that it is a kind of configuration rule (see, e.g., Reiff-Marganiec [47] in this book). The difference is that a feature typically is provisioned statically by a service provider, while a policy is intended to be defined dynamically by a user at run-time. Reiff-Marganiec [47] does not elaborate on the structure of the underlying communications layer of his policy architecture. It would be interesting research to extend our work to dynamically configured policies.

Legacy systems pose a challenge for the application of our ideas on requirements structuring. We proposed concrete improvements for the encapsulation of the user interface in the Intelligent Network [31], see the start of Sect. 6. But a general prerequisite is that rigorous requirements are documented explicitly. Already this can be difficult for a legacy system. However, the current migration to Voice over IP now offers the chance to conceive the requirements for such new systems as a family from the start.

A requirements module is a useful abstraction of the family of requirements. This returns us to our observation at the recent Feature Interaction Workshop (see the end of Sect. 2.2): it is important to consider the abstract purposes, not only the concrete mechanisms. Formulating good common abstractions explicitly is crucial.

Common abstractions need a domain with bounded change. These bounds can be hard to determine in the telephony domain. Take the example of the UMTS mobile network a few years ago. Everybody in the field would have agreed vigorously that the bandwidth downstream should be much higher than

upstream. Today, many envisioned services require a symmetric bandwidth distribution.

A prerequisite for any information hiding approach is our ability to predict the likeliness of changes to some degree. This holds both for information hiding in design and for information hiding in requirements. For requirements, we must put the most stable properties at the bottom of the requirements dependency partial order, and those most likely to change at the top. We don't know how to prepare for completely unanticipated changes.

References

1. Daniel Amyot and Luigi Logrippo, editors. "Feature Interactions in Telecommunications and Software Systems VII". IOS Press, Amsterdam (June 2003).
2. Muffy Calder and Evan Magill, editors. "Feature Interactions in Telecommunications and Software Systems VI". IOS Press, Amsterdam (May 2000).
3. Kristofer Kimbler and L. G. Bouma, editors. "Feature Interactions in Telecommunications and Software Systems V". IOS Press, Amsterdam (September 1998).
4. Petre Dini, Raouf Boutaba, and Luigi Logrippo, editors. "Feature Interactions in Telecommunication Networks IV". IOS Press, Amsterdam (June 1997).
5. Kong Eng Cheng and Tadashi Ohta, editors. "Feature Interactions in Telecommunications III". IOS Press, Amsterdam (1995).
6. L. G. Bouma and Hugo Velthuijsen, editors. "Feature Interactions in Telecommunications Systems". IOS Press, Amsterdam (1994).
7. ITU-T. "Q.12xx-Series Intelligent Network Recommendations" (2001).
8. James J. Garrahan, Peter A. Russo, Kenichi Kitami, and Roberto Kung. Intelligent Network overview. *IEEE Commun. Mag.* **31**(3), 30–36 (March 1993).
9. José M. Duran and John Visser. International standards for Intelligent Networks. *IEEE Commun. Mag.* **30**(2), 34–42 (February 1992).
10. Stephen Gilmore and Mark Ryan, editors. "Proc. of Workshop on Language Constructs for Describing Features", Glasgow, Scotland (15–16 May 2000). ESPRIT Working Group 23531 – Feature Integration in Requirements Engineering.
11. Stephen Gilmore and Mark D. Ryan, editors. "Language Constructs for Describing Features". Springer (2001).
12. Shmuel Katz. A superimposition control construct for distributed systems. *ACM Trans. Prog. Lang. Syst.* **15**(2), 337–356 (April 1993).
13. ITU-T, Recommendation Q.1203. "Intelligent Network – Global Functional Plane Architecture" (October 1992).
14. ITU-T, Recommendation Q.1224. "Distributed Functional Plane for Intelligent Network Capability Set 2: Parts 1 to 4" (September 1997).
15. Hugo Velthuijsen. Issues of non-monotonicity in feature-interaction detection. In Cheng and Ohta [5], pages 31–42.
16. E. Jane Cameron, Nancy D. Griffeth, Yow-Jian Lin, et al.. A feature interaction benchmark in IN and beyond. In Bouma and Velthuijsen [6], pages 1–23.
17. Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Comp. Networks* **41**, 115–141 (2002).

18. Pamela Zave. Feature disambiguation. In Amyot and Logrippo [1], pages 3–9.
19. Tom Gray, Ramiro Liscano, Barry Wellman, Anabel Quan-Haase, T. Radhakrishnan, and Yongseok Choi. Context and intent in call processing. In Amyot and Logrippo [1], pages 177–184.
20. Jan Bredereke. On preventing telephony feature interactions which are shared-control mode confusions. In Amyot and Logrippo [1], pages 159–176.
21. David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM* **15**(12), 1053–1058 (1972). Reprinted in [48].
22. David M. Weiss. Introduction to [21]. In Hoffman and Weiss [48], pages 143–144.
23. David Lorge Parnas, Paul C. Clements, and David M. Weiss. The modular structure of complex systems. *IEEE Trans. Softw. Eng.* **11**(3), 259–266 (March 1985). Reprinted in [48].
24. Kathryn Heninger Britton, R. Alan Parker, and David L. Parnas. A procedure for designing abstract interfaces for device interface modules. In "Proc. of the 5th Int'l. Conf. on Software Engineering – ICSE 5", pages 195–204 (March 1981). Reprinted in [48].
25. David Lorge Parnas. Designing software for ease of extension and contraction. *IEEE Trans. Softw. Eng.* **SE-5**(2), 128–138 (March 1979). Reprinted in [48].
26. David Lorge Parnas. On the design and development of program families. *IEEE Trans. Softw. Eng.* **2**(1), 1–9 (March 1976). Reprinted in [48].
27. David M. Weiss and Chi Tau Robert Lai. "Software Product Line Engineering – a Family-Based Software Development Process". Addison Wesley Longman (1999).
28. Reidar Conradi and Bernhard Westfechtel. Version models for software configuration management. *ACM Comput. Surv.* **30**(2), 232–282 (June 1998).
29. ITU-T, Recommendation Q.1223. "Global Functional Plane for Intelligent Network Capability Set 2" (September 1997).
30. Jan Bredereke. Maintaining telephone switching software requirements. *IEEE Commun. Mag.* **40**(11), 104–109 (November 2002).
31. Jan Bredereke. Avoiding feature interactions in the users' interface. In Kimbler and Bouma [3], pages 305–317.
32. Steven P. Miller. Specifying the mode logic of a flight guidance system in CoRE and SCR. In "Second Workshop on Formal Methods in Software Practice", Clearwater Beach, Florida, USA (4–5 March 1998).
33. Steven P. Miller and Karl F. Hoech. Specifying the mode logic of a flight guidance system in CoRE. Technical Report WP97-2011, Rockwell Collins, Inc., Avionics & Communications, Cedar Rapids, IA 52498 USA (November 1997).
34. Stuart R. Faulk, Jr. James Kirby, Lisa Finneran, and Assad Moini. Consortium requirements engineering guidebook. Tech. Rep. SPC-92060-CMC, version 01.00.09, Software Productivity Consortium Services Corp., Herndon, Virginia, USA (December 1993).
35. David Lorge Parnas and Jan Madey. Functional documents for computer systems. *Sci. Comput. Programming* **25**(1), 41–61 (October 1995).
36. Marcel Mampaey and Alban Couturier. Using TINA concepts for IN evolution. *IEEE Commun. Mag.* **38**(6), 94–99 (June 2000).
37. C. Abarca et al.. Service architecture. Deliverable, TINA-Consortium, URL http://www.tinac.com/ (16 June 1997). Version 5.0.
38. Michael Jackson and Pamela Zave. Distributed feature composition: A virtual architecture for telecommunications services. *IEEE Trans. Softw. Eng.* **24**(10), 831–847 (October 1998).

39. Gregory W. Bond, Eric Cheung, K. Hal Purdy, Pamela Zave, and J. Christopher Ramming. An open architecture for next-generation telecommunication services. *ACM Transactions on Internet Technology* **4**(1) (February 2004). *To appear.*
40. Israel Zibman et al.. Minimizing feature interactions: an architecture and processing model approach. In Cheng and Ohta [5], pages 65–83.
41. Debbie Pinard. Reducing the feature interaction problem using an agent-based architecture. In Amyot and Logrippo [1], pages 13–22.
42. Jan Bredereke. A tool for generating specifications from a family of formal requirements. In Myungchul Kim, Byoungmoon Chin, Sungwon Kang, and Danhyung Lee, editors, "Formal Techniques for Networked and Distributed Systems", pages 319–334. Kluwer Academic Publishers (August 2001).
43. Jan Bredereke. Families of formal requirements in telephone switching. In Calder and Magill [2], pages 257–273.
44. Clemens Fischer. Combination and implementation of processes and data: from CSP-OZ to Java. PhD thesis, report of the Comp. Sce. dept. 2/2000, University of Oldenburg, Oldenburg, Germany (April 2000).
45. Clemens Fischer. CSP-OZ: a combination of Object-Z and CSP. In Howard Bowman and John Derrick, editors, "Formal Methods for Open Object-Based Distributed Systems (FMOODS'97)", volume 2, pages 423–438. Chapman & Hall (July 1997).
46. Jan Bredereke. "genFamMem 2.0 Manual – a Specification Generator and Type Checker for Families of Formal Requirements". University of Bremen (October 2000). URL http://www.tzi.de/~brederek/genFamMem/.
47. Stephan Reiff-Marganiec. Policies: Giving users control over calls. In this book.
48. Daniel M. Hoffman and David M. Weiss, editors. "Software Fundamentals – Collected Papers by David L. Parnas". Addison-Wesley (March 2001).

Detecting Feature Interactions: How Many Components Do We Need?

Muffy Calder and Alice Miller

Department of Computing Science
University of Glasgow
Glasgow, Scotland
{muffy,alice}@dcs.gla.ac.uk

Abstract. Features are a structuring mechanism for *additional* functionality, usually in response to changing requirements. When several features are invoked at the same time, by the same, or different components, the features may not interwork. This is known as *feature interaction*. We employ a property-based approach to feature interaction detection: this involves checking the validity (or not) of a temporal property against a given system model. We use the logic LTL for temporal properties and the model-checker Spin to prove properties.
To gain any real insight into feature interactions, it is important to be able to infer properties for networks of *any* size, regardless of the underlying communication structure. We present an inference mechanism based on abstraction. The key idea is to model-check a system consisting of a constant number (m) of components together with an *abstract* component representing any number of other (possibly featured) components. The approach is applied to two systems with communication which is peer to peer and client server. We outline a proof of correctness in both cases.
The techniques developed here are motivated by feature interaction analysis, but they are also applicable to reasoning about networks of other types of components with suitable notions of data abstraction.

1 Introduction

Features are a structuring mechanism not dissimilar from objects and agents. For example, services are built up from a number of feature components. However, a major philosophical difference is that feature components are usually *additional* to a core body of software, often they are a response to new, or changing requirements. Typically, features are added incrementally, at various stages in the life cycle, usually by different developers, and often cutting across object boundaries. So when deployed, while each feature functions well on its own, they may not *interwork*. Namely, when several features are added to a service, or services are composed with each other there may be behavioural incompatibilities or modifications. This is known as the *feature interaction* problem [3, 13, 21, 4, 1]. While feature interactions are not necessarily undesirable (the addition of any feature to the base system is an interaction!), we need at least to know of their existence, before determining desirability or resolution. This is known as *detection*,

M.D. Ryan et al. (Eds.): Objects, Agents, and Features, LNCS 2975, pp. 45–66, 2004.

the subject of this paper. Although traditionally applied to telecommunications systems, the concept of feature interaction can equally apply to any distributed system in which features are offered.

We consider modelling features and analysing feature interactions in two different paradigms: a *telecommunications* system and an *email* system. Both systems consist of a network of basic components with different sets of features enabled. But the two paradigms represent different communication structures. The former is peer to peer, whereas the latter is client server.

In the first case, our model is based on the specification described more fully elsewhere [6] and follows the IN (*Intelligent Networks*) distributed functional plane [19]. In the second case, our model is based on a specification also described in more detail elsewhere [8], and is derived from Hall's email model [17].

To gain any real insight into the feature interaction problem, it is important to be able to infer properties for networks of *any* size, regardless of the underlying communication structure. But this is an example of the *parameterised model checking problem* (PMCP) which is, in general undecidable [2]. However, in some subclasses of systems the PMCP is decidable.

Our goal is to develop, in both paradigms,

- an interaction analysis which is *fully automated*, based on model-checking, and
- techniques to *infer* results about systems consisting of *any* number of components.

In the next section we introduce the concept of feature interaction, giving examples in the context of a telephone system and an email system. We explain the role of configurations and the novelty of our approach. In section 3 we describe the two network architectures that we will be considering. In sections 4 and 5 we give an overview of a basic service and feature behaviour in a telephony system and an email system respectively and in section 6 we provide a brief summary of the Promela implementation in each case.

In section 7 we give a brief overview of model checking and the model-checker Spin. We describe how model checking is used to perform feature interaction analysis on small, fixed size models of our examples and give the results. In section 8 we define PMCP and describe our solution, an abstraction technique. We apply it to our two example systems and give an outline of a proof of correctness. In section 9 we discuss our approach in the context of feature interaction analysis. Conclusions are in section 10.

We note that we have presented our basic Promela models and discussed feature interaction analysis for both paradigms elsewhere [6, 8]. In addition we have discussed our generalisation approach for the telecommunications example in [7] and for the email example in [8]. However, we have not compared the results for the two different communications architectures or provided any proof of the generalisation results in any previous publication.

The techniques presented here are motivated by feature interaction analysis, and illustrated in that context. However, they are, in principle, applicable to reasoning about about networks of other types of components such as objects

and agents. The only requirement is for suitable data abstractions and characterisations of the of observable behaviour of sets of components.

2 Feature Interaction

2.1 Feature Interactions in Telephony

Feature interaction detection in telecommunications has been the topic of intense research over the last decade [5]. In a telephone system, control of the progress of calls is provided by a (software) service at an exchange (a *stored program control* exchange). This software must respond to events such as handset on or off hook, as well as sending control signals to devices and lines such as ringing tone or line engaged. A *feature* is additional functionality, for example, a *call forwarding capability*, or *ring back when free*; a user is said to *subscribe* to or *invoke* a feature.

An example of an interaction is the following. Suppose a user subscribes to *call waiting* (CW) and *call forward when busy* (CFB) and is engaged in a call. What happens when there is a further incoming call? If the call is forwarded, then the CW feature is clearly compromised. If the subscriber receives the call waiting signal, then the CFB is compromised. In either case, the subscriber will not have his/her expectations met. This is an example of a single component (SC) interaction – the conflicting features are subscribed to by a single user. More subtle interactions can occur when more than one user/subscriber are involved, these are referred to as multiple component (MC) interactions. Consider when user A subscribes to *originating call screening* (OCS), with user C on the screening list, and user B subscribes to CFB to user C. If A calls B, and the call is forwarded to C, as prescribed by B's CFB, then A's OCS is compromised. If the call is not forwarded, then we have the converse. These kind of interactions can be particularly difficult to detect (and resolve), since different features are activated at different stages of a call.

2.2 Feature Interactions in Email

The problem of feature interaction in email was first suggested and investigated by Hall [17], who presented a systematic methodology based on simulation and formal test coverage. An example of an interaction is the obvious (desirable) interaction between an encryption feature and a decryption feature. However, a more subtle interaction arises between a filtering feature and an autoresponse feature. Suppose that a client A filters messages from client B, and that client A also has the autoresponse feature. If a message from B arrives at A, will the message be simply discarded, or will an automatic response be sent back to B? In either event, one feature is compromised.

2.3 Configurations for Feature Interaction Detection

Feature interaction detection involves examining scenarios, e.g. component A with features f_1 and f_2 performs some action which affects components B and

C with features f_3 and f_4 respectively. But in order to detect scenarios, one has to first determine the *configuration*: the number of components, and the enabled features for each component.

Nearly all analysis in the literature vary only one aspect of the configuration: the enabled features. They do not vary the number of components, but rather obtain results for a specific system of *fixed* size, i.e. for a system consisting of a *fixed* number of components. The results are (informally) assumed to hold for the general case, but there is no proof of such a generalisation.

In this paper, we first summarise feature interaction results obtained by model checking for systems of fixed size in both the telephone example and the email example. Second, we expand on the generalisation approach suggested in [7] and [8], to generalise our feature interaction results to networks of arbitrary size.

Our analysis is *pairwise*, known as 2-way interaction analysis. While at first sight this may appear limiting, empirical evidence suggests that 3-way interactions that are not detectable as a 2-way interaction are exceedingly rare [22].

2.4 Property Based Approach

A property based approach to feature interaction detection assumes a formal model of the entire system and a given set of properties (usually temporal) associated with the features. Two features are said to *interact* if a property that holds for the system when only one of the features is present, does not hold when both features are present. For features f_1 and f_2 we define feature interaction as follows:

Definition 1. *Let $\mathcal{M}$ be the model of a system of N components in which neither f_1 or f_2 are present and $\mathcal{M}(f_1)$, $\mathcal{M}(f_2)$ and $\mathcal{M}(f_1 \cap f_2)$ models in which only f_1, only f_2 and both f_1 and f_2 have been added respectively. If ϕ_1 and ϕ_2 are properties that define f_1 and f_2 respectively then f_1 and f_2 are said to interact if $\mathcal{M}(f_1) \models \phi_1$ but $\mathcal{M}(f_1 \cap f_2) \not\models \phi_1$; or $\mathcal{M}(f_2) \models \phi_2$ but $\mathcal{M}(f_1 \cap f_2) \not\models \phi_2$.*

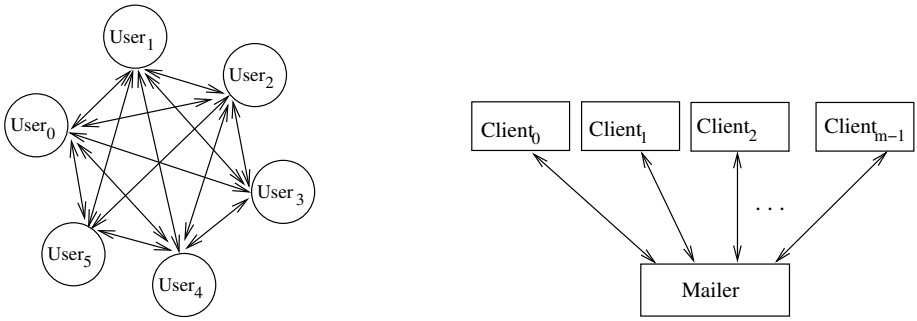
Note that this definition is relatively high-level, it does not contain details of the configuration. Thus it does not distinguish between SC and MC interactions. When we report on results later (section 7) we will make this distinction. Note also that this analysis will only reveal interactions that exist with respect to the particular properties ϕ_1 and ϕ_2. For complete analysis it may be necessary to perform analysis for a suite of properties for each feature, or to conjoin properties.

3 Communication within Telephone and Email Systems

The two communication mechanisms we consider are illustrated in Figure 1.

In peer to peer communication every member of a network can communicate with every other member of the network.

An example of such a network is an (unfeatured) basic telecommunications system. In our system, communication between call components (*Users*) takes



(a) Peer to peer (b) Client server

Fig. 1. Telephone and email network communication

place via channels. There is one channel associated with each user. Each channel has capacity for at most one message: a pair consisting of a channel name (the other party in the call) and a status bit (the status of the connection). Figure 1 (a) illustrates communication channels within a telephone system with 6 User components.

The email system is an example of a system which uses client server communication. This system consists of a number of *clients* and one server, in this case the *mailer* component. Figure 1 (b) illustrates an email system for m Client components. In our email system, each client has a unique mail address. Clients send mail messages, addressed to other clients (or themselves) to the mailer; the mailer delivers mail messages to clients. Communication between client and server is asynchronous. Therefore, mail messages are not necessarily received by clients in the (global) order in which they were sent, but local temporal ordering is maintained, i.e. if client A sends messages 1 and 2 to client B, in that order, then client B will always receive message 1 before message 2 (though it may receive other messages in between).

4 Basic Telephone System and Features

Figure 2 is a high-level, abstract automaton for the basic call service behaviour (note the full implementation is somewhat more complicated, for example, some states (e.g. *unobtainable*) have been omitted). States to the left of the idle state represent *terminating* behaviour, states to the right represent *originating* behaviour. Transitions between states are triggered by *user-initiated* events at the terminal device, such as (handset) on and (handset) off, or by *communication* events on shared channels. We have excluded some trivial behaviour from the automaton. For example, it is possible to perform a dial event (with no effect) from most states. Note also that while the state preidle is an important detail of the implementation (where local and global variables are reset), it does not play a part in the observable behaviour of a call component.

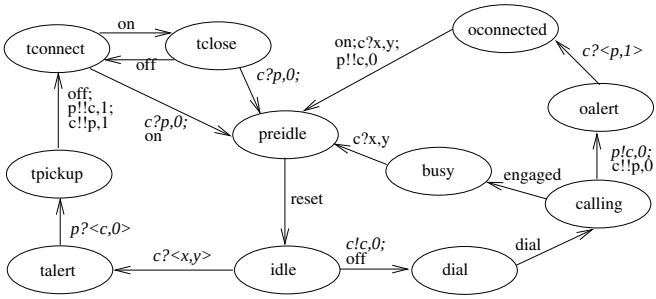


Fig. 2. Basic Call - States and Events

Originating and terminating automata can affect each other's behaviour through communication via (shared) channels. In the automaton, the channels are referred to as c, for the channel associated with that component, and p, for the channel associated with the partner component. In the originating side of the automaton, p is chosen *non-deterministically*. Otherwise, p is determined by the nature of incoming messages. We use the notation $c!x, y$ to denote *write* the value (x, y) to the channel c, $c!!x, y$ to denote *overwrite* the channel c with (x, y), $c? < x, y >$ to denote *poll* or non-destructively read value (x, y) from channel c, and $c?x, y$ to denote *destructively read* value (x, y) from channel c. When the value may be arbitrary, we use variables x and y; otherwise we use the actual constants required, e.g. 0,1, p, etc. If a read or write statement is written in italics it implies a *condition*. The appropriate action should be taken if and when the relevant channel is not empty/full. When there are two transitions from a particular state, one of which has a condition labelling it (e.g. from *calling*) the italicised transition should be taken if the condition holds.

A call component is not connected to, or attempting to connect to, any other call component when its associated communication channel is empty, When a communication channel is not empty, then the associated call component is *engaged in a call*, but not necessarily connected to another user. The interpretation of messages is described more comprehensively in [7].

4.1 Features of the Telephone System

Consider a set of 7 features to be added to the basic call, with the following properties.

CFU – call forward unconditional. Assume that $User[j]$ forwards to $User[k]$. If $User[i]$ rings $User[j]$ then a connection between $User[i]$ and $User[k]$ will be attempted before $User[i]$ hangs up.
CFB – call forward when busy. Assume that $User[j]$ forwards to $User[k]$. If $User[i]$ calls $User[j]$ when $User[j]$ is busy then a connection between $User[i]$ and $User[k]$ will be attempted before $User[i]$ hangs up.
OCS – originating call screening. Assume that $User[i]$ has $User[j]$ on its screening list, $i \neq j$. No connection from $User[i]$ to $User[j]$ is possible.

ODS – originating dial screening. Assume that $User[i]$ has $User[j]$ on its screening list, $i \neq j$. $User[i]$ may not dial $User[j]$.
TCS – terminating call screening. Assume that $User[i]$ has $User[j]$ on its screening list, $i \neq j$. No connection from $User[j]$ to $User[i]$ is possible.
RBWF – ring back when free. Assume that $User[i]$ has RBWF. If $User[i]$ has requested a ringback to $User[j]$, $i \neq j$, (and not subsequently requested a ringback to another user) and subsequently $User[i]$ is idle when $User[i]$ and $User[j]$ are both free (and they are still free when $User[i]$ is no longer idle) then $User[i]$ will hear the ringback tone.
OCO – originating calls only. Assume that $User[j]$ has OCO. No connection from $User[i]$ to $User[j]$ is possible.
TCO – terminating calls only. Assume that $User[j]$ has OCO. No connection from $User[j]$ to $User[i]$ is possible.
RWF – return when free. Assume that $User[j]$ has RWF. If $User[i]$ calls $User[j]$ when $User[j]$ is busy ($i \neq j$), then $User[i]$ will hear the *ret_alert* tone and $retnum[i]$ will be set to j (before $User[i]$ returns to the idle state).

The logic we use to formalise these properties is LTL – linear temporal logic. This logic has temporal operators [] (always), $\langle\rangle$ (eventually), X (next) and U (weak until), the only path operator is (implicit) universal quantification. Conjunction is denoted by $\wedge$. We do not give full details of all the LTL here, but give one example the formula for RBWF:

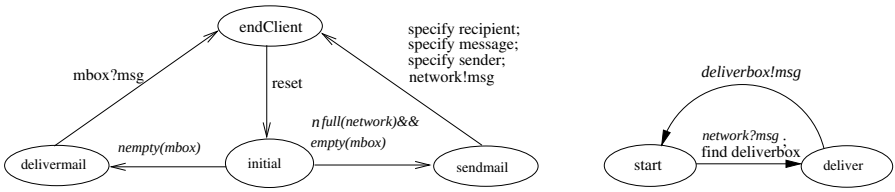
$$[]\neg((p \wedge q \wedge r \wedge s) \wedge ((p \wedge q \wedge r \wedge s)U((p \wedge (\neg q) \wedge r) \wedge ((\neg t)Uq))))$$

where $p = (rgbknum[i] == j)$, $s = (len(chan\_name[i]) == 0)$, $q = (User[i]$ $@idle)$, $r = (len(chan\_name[j]) == 0)$, and $t = (network\_event[i] == ringbackev)$.

The propositions *p,q* etc. refer to the state of internal variables, these should be self-explanatory.

5 Basic Email Service and Features

High level, abstract automata for the client and mailer components are given in figure 3. Note that in these figures, some transitions are again labelled by conditions, e.g. in figure 3(a) a transition from *initial* to *sendmail* is only possible if the channel *mbox* is empty and the channel *network* is not full. Local and global variables are updated at various points; most variable assignments (apart from *reset*, in which all local variables are reset to their original values) are omitted from the diagrams. In order to avoid continuous blocking of the *network* channel, Client behaviour must occur in one of two indivisible loops. If the Client's mailbox is non-empty, the Client can send a message in which case variables are reset, and the Client returns to the *initial* state. Otherwise, if the *network* channel is not full and the Client's mailbox is empty, the Client may send a message via the *network* channel. Again variables are reset and the Client returns to the *initial* state. Unlike the automata of figures 2 and 3(b) an indivisible global transition (or *atomic* statement – see section 6) is represented



(a) Client component with mailbox *mbox* (b) Mailer component

Fig. 3. Email components

by one of these loops. As a result, in figure 3(a) notice that the write statement from the *sendmail* state to the *endClient* state is unitalicised. This is because this transition happens *immediately* after the transition from *initial* to *sendmail*, at which point it is established that the *network* channel is not full. Thus a write to this channel will be always be enabled. In figures 2 and 3(b) on the other hand, each transition between the *abstract* states (*idle*, *dial* etc. or *start* and *deliver* respectively) is an indivisible global transition.

5.1 Features of the Email System

Hall's email model [17] included a suite of 10 features. Consider 7 of these features.

Encryption. If $Client[i]$ has encryption on, then if $Client[j]$ receives a message whose sender is $Client[i]$, then the message will be encrypted.
Decryption. If $Client[i]$ has decryption on, then all messages received by $Client[i]$ will have been decrypted.
Autoresponse. If $Client[i]$ has autorespond on, then if $Client[j]$ sends a message to $Client[i]$, and $Client[j]$ hasn't already received an automatic response from $Client[i]$, then $Client[j]$ will eventually receive a reply from $Client[i]$. Alternatively, $Client[i]$ eventually stops sending messages because network can't be accessed.
Forwarding. If $Client[i]$ forwards messages to $Client[j]$, then it is possible for $Client[j]$ to receive messages not addressed to $Client[j]$ (or to the default value M).
Filtering. If Mailer filters messages from $Client[i]$ to $Client[j]$ then it is not possible for $Client[j]$ to receive a message from $Client[i]$.
Mailhost. If mailhost is on and $Client[i]$ is a non-valid name $Client[j]$ sends a message to $Client[i]$ then, if $i \neq j$, $Client[j]$ will eventually receive a message from *postmaster*.
Remail. Suppose that $Client[i]$ has the remailer feature and has a pseudonym of k. If $Client[i]$ sends a message to $Client[j]$, then $Client[j]$ will eventually receive a message from k. Also, if $Client[j]$ sends a message to k, then if $j \neq i$, $Client[i]$ will eventually receive a message from $Client[j]$.

Again, we do not give the LTL for all properties, the first five are given in [8]. The LTL for *mailhost* and *remail* are as follows:

Mailhost: $[](p- > \langle\rangle q)$ assuming $i \neq j$, $p = (last\_sent\_from[j]\_to == i)$ and $q = (last\_del\_to[j]\_from == pseud(i))$.

Remail: the conjunction of:

$[](((\neg p) \wedge X(p))- > X(\langle\rangle q))$, where $p = (last\_sent\_from[i]\_to == j)$ and $q = (last\_del\_to[j]\_from == pseud(i))$, and

$[](((\neg p) \wedge X(p))- > X(\langle\rangle q))$, assuming $i \neq j$, $p = (last\_sent\_from[j]\_to == pseud(i))$ and $q = (last\_del\_to[i]\_from == j)$.

Again, the propositions refer to internal variables and should be self-explanatory.

6 Implementation in Promela

Both example systems have been implemented in Promela, the source language for the model-checker Spin. Promela is an imperative, C-like language with additional constructs for non determinism, asynchronous and synchronous communication, dynamic process creation, and mobile connections, i.e. communication channels can be passed along other communication channels.

In the telecomms example, each call component (see figure 2) is an instantiation of the (parameterised) proctype *User*. Similarly, in the email model, each *Client* component (see figure 3(a)) and the *Mailer* component (see figure 3(b)) are instantiations of a *Client* and *Network_Mailer* proctype. Code relating to each *abstract* state (e.g. *idle*, *dial* etc. and *deliver*, *start* etc.), is contained within an atomic statement. The atomic statements cannot block because there are no *read* or *write* statements from a possibly empty or full channel contained within the atomic statement, except possibly at the beginning of the statement. Thus the statement is either unexecutable, or the whole statement will be executed as one. This ensures that every global transition in the resulting model involves a change in the abstract state of one component.

In both examples, features are added to the code by way of *inline* statements (a Promela procedure mechanism). In the telecomms example the *feature_lookup* inline encapsulates centralised intelligence about the state of calls, i.e. what is known as single point call control. Calls at pertinent places in the code result in different behaviour according to whether relevant features are *switched on*. In the email example, a separate inline is included in the code for each feature. In most cases, feature implementation merely involves calls to these inlines to determine if the relevant feature is switched on. In general, the presence of the features simply results in additional transitions or steps during one or more of the abstract states of the client or mailer components. The exception is autorespond, because this feature involves both *reading* – a message from a client channel, and *writing* – a message to the network channel. Both events are potentially blocking, hence cannot take place within one atomic step. Therefore, to implement this feature, we add an additional data structure to indicate whether or not a client

requires to send an autoresponse. We enhance the *initial* state to include the possibility that an autoresponse message needs to be sent, and give priority to this over any other event.

7 Model Checking

Model checking is an technique for verifying finite state systems. Systems are specified using a modelling language and the model – or *Kripke structure* [9] associated with this specification is checked to verify given temporal properties. In this section we give a a brief explanation of Spin, followed by a formal definition of model checking and results of feature interaction analysis for fixed sized systems.

7.1 Reasoning with SPIN

Spin [18] is the bespoke model-checker for Promela and provides several reasoning mechanisms: assertion checking, acceptance and progress states and cycle detection, and satisfaction of temporal properties, expressed in LTL.

Spin translates each component defined in the Promela specification into a finite automaton and then computes the asynchronous interleaving product of these automata to obtain the global behaviour of the concurrent system. This interleaving product is essentially a Kripke structure (see below), describing the behaviour of the system. It is this Kripke structure to which we refer when we talk about the *model* of our system. The set of states of this Kripke structure is referred to as the *state-space* of the model.

7.2 Kripke Structures and Model-Checking

Definition 2. *Let AP be a set of atomic propositions. A Kripke structure over AP is a tuple* $\mathcal{M} = (S, S_0, R, L)$ *where* S *is a finite set of states,* S_0 *is the set of initial states,* $R \subseteq S \times S$ *is a transition relation and* $L : S \to 2^{AP}$ *is a function that labels each state with the set of atomic propositions true in that state.*

For a given model $\mathcal{M}$, and temporal property ϕ, model checking allows us to show that $\mathcal{M} \models \phi$. This is known as the *model checking problem.*

7.3 Feature Interaction Analysis for Models of Small Size

We give our feature interaction detection results for the two example systems.

Tables 1 and 2 indicate the feature interactions obtained for a telephone system and an email system with a small number of User/Client components. A x denotes no interaction, S and M denote single and multiple component interactions, respectively. In this section we limit ourselves to at most 4 components. Indeed, unless checking for MC interactions between a pair of filtering or forward features, 3 suffice. However we show in section 9 that for complete analysis it

would be necessary to consider 5 or more components in some situations. Note that properties relating to forwarding and (in the email system) mailhost assume that $i \neq j$. It is important that our results are analysed to ensure that no false interactions are recorded. For example, although the filtering property is violated when features $filter[0] = 1$ and $filter[0] = 2$ are both selected, this is due to the fact that we only allow screening lists to have length 1 (so the second feature *overrides* the first). Similarly, we do not record an SC interaction for two forwarding features.

Table 1. Feature interaction results for the telephone example

	CFU	CFB	OCS	ODS	TCS	RBWF	OCO	TCO	RWF
CFU	M	S,M	S,M	×	M	×	×	×	×
CFB	S,M	M	S,M	×	S,M	×	×	×	×
OCS	×	×	×	×	×	×	×	×	×
ODS	M	M	×	×	×	×	×	×	×
TCS	×	×	×	×	×	×	×	×	×
RBWF	×	×	×	×	×	×	×	×	×
OCO	×	×	×	×	×	×	×	×	×
TCO	×	×	×	×	×	×	×	×	S
RWF	×	×	×	×	×	×	×	×	×

Table 2. Feature interaction results for the email example

	Encrypt	Decrypt	Filter	Forward	Autoresp	Mailhost	Remail
Encrypt	×	S,M	×	×	×	×	×
Decrypt	×	×	×	×	×	×	×
Filter	×	×	×	M	×	×	×
Forward	×	×	×	M	×	×	×
Autoresp	×	×	S,M	S,M	×	S,M	S,M
Mailhost	×	×	S	S	×	M	M
Remail	×	×	S	S,M	×	S,M	M

7.4 Use of Perl Scripts

For each pair of features, set of feature parameters, associated property and set of property parameters, a relevant model needs to be individually constructed to ensure that only relevant variables are included and set. For each example system, we have developed two Perl scripts, for automatically configuring the model and for generating model-checking runs. These scripts greatly reduce the time to prepare each model and the scope for errors. The results reported above were obtained using these scripts (running overnight). It is important to note that a certain amount of simple symmetry reduction is incorporated within the

Perl script to avoid repeating runs of configurations which are identical up to renaming of components.

8 Any Number of Components

An obvious limitation of the model checking approach is that only finite-state models can be checked for correctness. Sometimes however we wish to prove correctness (or otherwise) of *families* of (finite-state) systems. That is to show that, if $\mathcal{M}_N = \mathcal{M}(p_0||p_1||\ldots||p_{N-1})$ is the model of a system of N concurrent instantiations of a parameterised component p, then $\mathcal{M}_N \models \phi$ for all $N \geq 1$.

This is known as the *parameterised Model Checking problem* which is, in general, undecidable [2]. The verification of parameterized networks is often accomplished via theorem proving [25], or by synthesising network invariants [10, 23, 26]. Both of these approaches require a large degree of ingenuity.

In some cases it is possible to identify subclasses of parameterised systems for which verification is decidable. Examples of the latter mainly consist of systems of N identical components communicating within a ring topology [15, 16] or systems consisting of a family of N identical *user* components together with a *control* component, communicating within a star topology [24, 16, 20]. A more general approach [14] considers a general parameterised system consisting of several different classes of components.

One of the limitations of both the network invariant approach and the subclass approach is that it can only be applied to systems in which each component (contained in the set of size N) is completely independent of the overall structure of the system: adding an extra component (to this set) does not change the semantics of the existing components. A generalisation of data independence is used to verify arbitrary network topologies [12] by lifting results obtained for limited-branching networks to ones with arbitrary branching.

All of these methods fail when applied to asynchronously communicating components like ours, where components communicate asynchronously via shared variables.

We describe an abstraction technique that is applicable to asynchronously communicating components, namely it enables us to infer properties of a model of system of any size from properties of a finite *abstract* model. The key idea is to define an abstract component – an environment – which represents the (observable) behaviour of a number of components. For a given m, we then examine the behaviour of the system consisting of m (slightly modified) components and the abstract component. From this behaviour, we can infer the general behaviour. In the following, we apply the approach to our two examples, and outline a proof of correctness.

8.1 The *abstract* Model for the Telephone System

Let us first consider the telephone system. For any feature f, we say that f is *indexed* by $I_f = \{i_0, \ldots i_{r-1}\}$ if the feature relates to $User[i_0], \ldots, User[i_{r-1}]$.

For example if f is "$User[0]$ forwards calls to $User[3]$", then f is said to be indexed by 0 and 3. Similarly we say that a property ϕ is indexed by a the set I_ϕ where I_ϕ is the set of User ids associated with ϕ. For a (possibly empty) set of features $F = \{f_0 \dots f_{s-1}\}$ and property ϕ, we define the *complete index set* I of $\{\phi\} \cup F$, to be $I_{f_0} \cup \dots \cup I_{f_{s-1}} \cup I_\phi$.

Suppose that we have a system S of N telephone components (with or without features) where $S = p_0||p_1|| \dots ||p_{N-1}$ with associated model $\mathcal{M}_N = \mathcal{M}(S)$. For any $m \leq N$ we define an abstract system $abs_t(m)$ where

$$abs_t(m) = p'_0||p'_1|| \dots ||p'_{m-1}||Abstract_t(m) \text{ if } m < N$$

or

$$abs_t(m) = p'_0||p'_1|| \dots ||p'_{m-1} \text{ otherwise.}$$

(For simplicity we will assume from now on that $m < N$.) In this system, the p'_i, for $0 \leq i \leq m-1$ are *modified* User components and behave exactly the same as the original (*concrete*) components, p_i, $0 \leq i \leq m-1$ except that, for $0 \leq i \leq m-1$:

1. component p'_i no longer writes to (the associated channels of) any of the components $p_m, p_{m+1}, \dots, p_N$ (the *abstracted* components), but there is a non-deterministic choice whenever such a write would have occurred as to whether the associated channel is empty or full (thus, whether the write is enabled or not).
2. An initial call request from any *abstracted* component to p'_i now takes the form $(out\_channel, 0)$, regardless of which abstracted component initiated the call. When such a message arrives on p'_i's channel, p'_i may read it. Henceforth p'_i no longer reads from (the associated channels of) any of the abstracted components. Instead, p'_i makes a non-deterministic choice over the set of possible messages (if any) that could be present on such a channel.

The component $Abstract_t(m)$ encapsulates part of the observable behaviour of all of the abstracted components. The component $Abstract_t(m)$ has $id = m$ and an associated channel named *out_channel*. A call initiation from an abstracted component to a concrete component is replaced by a message of the form $(out\_channel, 0)$ from $Abstract_t(m)$ to the relevant channel (*zero*, *one* etc.) which is always possible, provided the channel is empty. Any other message passing from the abstracted components is now represented by the non-deterministic choice available to the modified components, as described above.

In particular, suppose that $S = p_0||p_1|| \dots ||p_{N-1}$ is a system of telephone components in which at least the features F are present. If ϕ is a property and only components $p_0, p_1, \dots, p_{m-1}$ are involved in the features F or in ϕ then, regardless of whether components $p_m, p_{m+1}, \dots, p_{N-1}$ have associated features or not (with some conditions attached), we will show that if ϕ holds for the model associated with $abs_t(m)$ (namely $\mathcal{M}_{abs_t(m)}$), then it holds for $\mathcal{M}(S)$. This case is illustrated in figure 4.

In fact, because there is a bidirectional correspondence between $\mathcal{M}_{abs_t(m)}$ and

$$\mathcal{M}(p'_{i_0}||p'_{i_1}|| \dots ||p'_{i_{m-1}}||Abstract_t(m'))$$

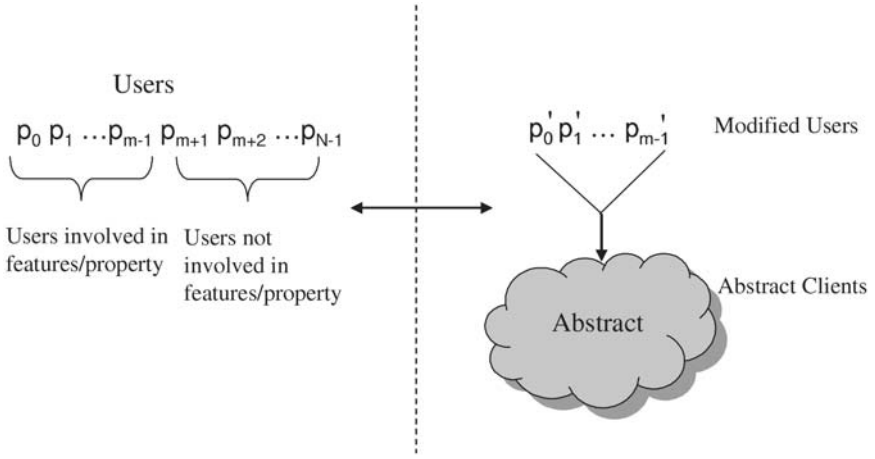


Fig. 4. Abstraction technique for N-User telephone model

where $\{i_0, i_1, \ldots, i_{m-1}\}$ is a subset of $\{0, 1, \ldots, N-1\}$ of size m and m' the smallest element of $\{0, 1, \ldots, N-1\} \setminus \{i_0, i_1, \ldots, i_{m-1}\}$, we can extend this result to all cases where the total index set has size m. Thus we show:

Theorem 1. *Let $S = p_0||p_1||\ldots||p_{N-1}$ be a system of telephone components in which at least the features F are present, and ϕ a property.*

1. *If the total index set of $F \cup \{\phi\}$ is $\{0, 1, \ldots, m-1\}$ then if components $p_m, p_{m+1}, \ldots, p_{N-1}$ do not have any of the features CFU, CFB or TCS, $\mathcal{M}_{abs_t(m)} \models \phi$ implies that $\mathcal{M}(S) \models \phi$.*
2. *If the total index set of $F \cup \{\phi\}$ is $\{i_0, i_1, \ldots, i_{m-1}\}$ and σ the permutation that maps j to i_j for $0 \leq j \leq m-1$ and m to m' where m' is the smallest element of $\{0, 1, \ldots, N-1\} \setminus \{i_0, i_1, \ldots, i_{m-1}\}$, then if components $p_{\sigma(m)}, p_{\sigma(m+1)}, \ldots, p_{\sigma(N-1)}$ do not have any of the features CFU, CFB or TCS, $\mathcal{M}_{abs_t(m)} \models \phi$ implies that $\mathcal{M}(S) \models \sigma(\phi)$.*

Application of this theorem, for example, allow us to infer that many of the results of Table 1 (for networks of size 3 or 4) scale up to networks of arbitrary size. Proof of the theorem is outlined below.

Note that the features CFU, CFB and TCS are the only features in our feature set whose presence in the *partner* of a User component affects the behaviour of the User itself (the host). As can be seen from the proof below, this is the reason that their presence in the abstracted components is disallowed.

An Outline of the Proof of Correctness of the Abstraction. We will assume throughout that the components $p_0, p_1, \ldots, p_{m-1}$ do not have any features other than those contained in the set F. The first stage of the proof of correctness of Theorem 1 involves the construction of a reduced model $\mathcal{M}_r^m$ via data abstraction [11] for any $m \leq N$. First we give some definitions:

Definition 3. *Let $X = \{x_0, x_1, \ldots, x_{l-1}\}$ denote a set of variables such that each variable x_i ranges over a set D_i. Then $D = D_0 \times D_1 \times \cdots \times D_{l-1}$ is called the domain of X. A set of abstract values $D' = D'_0 \times D'_1 \times \cdots \times D'_{l-1}$ is called an abstract domain of X if there exist surjections $h_0, h_1, h_2, \ldots, h_{l-1}$ such that $h_i : D_i \to D'_i$ for all $0 \leq i \leq l-1$. If such surjections exist they induce a surjection $h : D \to D'$ defined by*

$$h((x_0, x_1, \ldots, x_{l-1})) = (h_0(x_0), h_1(x_1), \ldots, h_{l-1}(x_{l-1})).$$

In the following definition (taken from [9]) data abstraction is used to define a reduced structure whose variables are defined over an abstract domain:

Definition 4. *Let $\mathcal{M} = (S, R, S_0, L)$ be a Kripke structure with set of atomic propositions AP and set of variables X with domain D. If D' is an abstract domain of X and h the corresponding surjection from D to D' then h determines a set of abstract atomic propositions AP'. Let $\mathcal{M}'$ denote the structure identical to $\mathcal{M}$ but with set of labels L' where L' labels each state with a set of abstract atomic propositions from AP'. The structure $\mathcal{M}'$ can be collapsed into a reduced structure $\mathcal{M}_r = (S_r, R_r, S_0^r, L'_r)$ where*

1. *$S_r = \{L'(s) | s \in S\}$, the set of abstract labels.*
2. *$s_r \in S_0^r$ if and only if there exists s such that $s_r = L(s)$ and $s \in S_0$.*
3. *$AP_r = AP'$.*
4. *As each s_r is a set of atomic propositions, $L_r(s_r) = s_r$.*
5. *$R_r(s_r, t_r)$ if and only if there exist s and t such that $s_r = L'(s)$, $t_r = L'(t)$, and $R(s, t)$.*

The following lemma (which is a restriction of a result proved in [11]) shows how we may use a reduced structure $\mathcal{M}_r$ to deduce properties of a structure $\mathcal{M}$.

Lemma 1. *If $\mathcal{M}$ and $\mathcal{M}_r$ are a Kripke structure and a reduced Kripke structure as defined in definition 4 then for any LTL property ϕ, $\mathcal{M}_r \models \phi$ implies that $\mathcal{M} \models \phi$.*

We do not give full details of our reduced model $\mathcal{M}_r^m$ here, but instead give a brief description of the abstract domains involved.

The abstract domains of local variables of components $p_m, p_{m+1}, \ldots, p_{N-1}$ are the trivial set $\{true\}$. In $\mathcal{M}_N$ all other variables, apart from those associated with channel names or contents, have domains equal to the set $\{0, 1, \ldots, N-1\}$ (the set of component ids). Each of these variables have abstract domains equal to the set $\{0, 1, \ldots, m-1\}$ and a surjection from the original domain D to the abstract domain D' is given by $h_1 : D \to D'$ where

$$h_1(x) = \begin{cases} x & \text{if } x < m, \\ m & \text{otherwise} \end{cases}$$

for all $x \in D$. In $\mathcal{M}$, the domains of channel variables such as $self$ and $partner$, consist of the set of channel names $name[0], name[1], \ldots, name[N-1]$ (where

$name[0]$, $name[1]$, etc. represent the channel names *zero*, *one*, etc.). The abstract domains for such variables is $name[0], name[1], \ldots, name[m]$ and the surjection h_2 is an obvious extension of h_1 above. Similarly abstract domains for the variables of contents of channels $name[0], name[1], \ldots, name[m-1]$ (a channel name and a status bit in each case) can be defined, and a surjection given in each case. The abstract domains for the variables of contents of channels $name[0], name[1], \ldots, name[m-1]$ are the trivial set.

From lemma 1 it follows that for any LTL property ϕ, $\mathcal{M}_r \models \phi$ implies that $\mathcal{M} \models \phi$.

The next stage of our proof involves showing that, for all $m \leq N$, $\mathcal{M}_r^m$ *simulates* $\mathcal{M}_{abs_t(m)}$. Again we provide some useful definitions:

Definition 5. *Given two structures $\mathcal{M}$ and $\mathcal{M}'$ with $AP \supseteq AP'$, a relation $H \subseteq S \times S'$ is a simulation relation between $\mathcal{M}$ and $\mathcal{M}'$ if and only if for all s and s', if $H(s, s')$ then*

1. $L(s) \cap AP' = L'(s')$
2. *For every state s_1 such that $R(s, s_1)$, there is a state s_1' with the property that $R'(s', s_1')$ and $H(s_1, s_1')$.*

If a simulation relation exists between structures $\mathcal{M}$ and $\mathcal{M}'$ we say that $\mathcal{M}'$ simulates $\mathcal{M}$ and denote this by $M \preceq M'$.

Lemma 2. *Suppose that $\mathcal{M} \preceq \mathcal{M}'$. Then for every LTL formula ϕ with atomic propositions in AP', $M' \models \phi$ implies $M \models \phi$.*

To prove that, for all $m \leq N$, $\mathcal{M}_r^m$ simulates $\mathcal{M}_{abs_t(m)}$, it is first necessary, for all $m \leq N$, to define a relation between the set of states of $\mathcal{M}_r^m$ (S_r^m say) and the set of states of $\mathcal{M}_{abs_t(m)}$ ($S_{a,t}^m$ say). Suppose V is the set of variables associated with $\mathcal{M}_N$ and V_r a reduced set of variables, such that V_r is identical to V except that the local and global variables associated with components $p_m, p_{m+1}, \ldots, p_{N-1}$ have been removed. The atomic propositions relating to $\mathcal{M}_N$ is the set $AP = \{x = y : x \in V \text{ and } y \in D(x)\}$, where $D(x)$ is the domain of x. Let us consider the alternative set of atomic propositions $AP' = \{x = y : x \in V_r \text{ and } y \in D'(x)\}$, where $D'(x)$ is the abstract domain of x. If we let L_r denote the labelling function associated with AP', then we can define a relation H between S_r^m and $S_{a,t}^m$ as follows: For $s \in S_r^m$ and $s' \in S_{a,t}^m$, $H(s, s')$ if and only if $L_r(s) = L_r(s')$.

To show that H is a simulation relation, it is necessary to show that for all $(s, s') \in H$, every transition from (s, s_1) in $\mathcal{M}_r^m$ is matched by a corresponding transition (s', s_1') in $\mathcal{M}_{abs_t(m)}$, where $(s_1, s_1') \in H$. Every transition in $\mathcal{M}_r^m$ either only involves a change to the global variables or involves a change to the value of the local variables of one of the (concrete) components. If the former is true, then the transition involves an initial message being placed on the channel of one of the (concrete) components $p_0, p_1, \ldots, p_{m-1}$ by one of the components $p_m, p_{m+1}, \ldots, p_{N-1}$. This transition is reflected in $\mathcal{M}_{abs_t(m)}$ by a transition involving the *Abstract* component in which a message is placed on the channel

of the concrete component. If t is a transition in $\mathcal{M}_r^m$ involving concrete component $p(i)$ then either t does not involve any component other than $p(i)$ or t involves only component p_i plus another concrete component or one or more of the following holds:

1. Component p_i is not currently in communication with another component and t involves a read from the channel of p_i as a result of the initiation of communication by one of the components $p_m, p_{m+1}, \ldots, p_{N-1}$.
2. Component p_i is currently in communication with one of the components $p_m, p_{m+1}, \ldots, p_{N-1}$ and t involves p_i reading a message from this component or
3. Component p_i is currently in communication with one of the components $p_m, p_{m+1}, \ldots, p_{N-1}$ and t involves a call to the *feature_lookup* function.

(Notice that a write to one of the components $p_m, p_{m+1}, \ldots, p_{N-1}$ does not involve a change of state, as the abstract domains associated with the channel contents of each of these components is trivial.)

Let $t = (s, s_1)$ and suppose that $H(s, s')$ for $s' \in S_{a,t}^m$. If $t = (s, s_1)$ does not involve any component other than p_i, or t involves p_i and another concrete component, there is clearly an identical transition $(s', s_1') \in \mathcal{M}_{abs_t(m)}$ such that $H(s, s')$.

If t involves a read from its channel of an initial message sent by one of the components $p_m, p_{m+1}, \ldots, p_{N-1}$ then t is reflected by a transition in $\mathcal{M}_{abs_t(m)}$ (p_i will still read such a message). However, if t involves any other read from one of the components $p_m, p_{m+1}, \ldots, p_{N-1}$, non-deterministic choice in p_i' (the corresponding component in $abs_t(m)$) ensures that an equivalent transition in $\mathcal{M}_{abs_t(m)}$ exists.

If p_i is currently involved in communication with one of the components $p_m, p_{m+1}, \ldots, p_{N-1}$ and t involves a call from p_i to the *feature_lookup* function then we must consider the cases of when components $p_m, p_{m+1}, \ldots, p_{N-1}$ have no associated features, and when they do have associated features, separately. (We have previously assumed that the components $p_0, p_1, \ldots, p_{m-1}$ do not have any features other than those contained in the set F.)

If components $p_m, p_{m+1}, \ldots, p_{N-1}$ have no associated features, then any guard g within the *feature_lookup* function that holds for a state in $\mathcal{M}_N$ (from which *feature_lookup* is called) will hold at the associated state in $\mathcal{M}_r^m$. If s is such a state and s' a state in $\mathcal{M}_{abs_t(m)}$ such that $H(s, s')$ then g holds at s' and it is clear that any transition t in $\mathcal{M}_r^m$ from s is reflected in $\mathcal{M}_{abs_t(m)}$.

However, if components $p_m, p_{m+1}, \ldots, p_{N-1}$ have associated features the situation is more difficult. Some guards within the *feature_lookup* inline depend only on whether the associated feature is present within the *host* component. Others depend on whether the feature is present within the *partner* component. Knowledge of the particular features that are present in the components $p_m, p_{m+1}, \ldots, p_{N-1}$ would be required to determine if the guards are true or not in the latter case. This scenario presents an open problem, which is beyond the scope of this paper. However, of our 9 features, guards relating to only 3 features depend on whether the partner component has the feature (namely CFU, CFB

and TCS). Therefore, under the conditions of Theorem 1, we can conclude that there is a simulation relation between $\mathcal{M}_r^m$ and $\mathcal{M}_{abs_t(m)}$.

From Lemma 2 we can conclude that, for any LTL property ϕ, if components $p_m, p_{m+1}, \ldots, p_{N-1}$ do not subscribe to features CFU, CFB or TCS then, if $\mathcal{M}_{abs_t(m)} \models \phi$ then $\mathcal{M}_r^m \models \phi$ and the first part of Theorem 1 follows from Lemma 1.

It is straightforward to show that there is a correspondence between $\mathcal{M}_{abs_t(m)}$ and

$$\mathcal{M}(p'_{i_0}||p'_{i_1}||\ldots||p'_{i_{m-1}}||Abstract_t(m'))$$

where $\{i_0, i_1, \ldots, i_{m-1}\}$ is a subset of $\{0, 1, \ldots, N-1\}$ of size m and m' the smallest element of $\{0, 1, \ldots, N-1\} \setminus \{i_0, i_1, \ldots, i_{m-1}\}$. The second part of Theorem 1 follows as a result of this correspondence.

8.2 The *abstract* Model for the Email System

Suppose that we have a system S of $N-1$ Client components and a Mailer component where $S = M_0||p_1||p_2||\ldots||p_{N-1}$ with associated model $\mathcal{M}_N = M_S$. For any $m \leq N$ we define an abstract system $abs_e(m)$ where

$$abs_e(m) = M'_0||p'_1||p'_2||\ldots||p'_{m-1}||Abstract_e(m) \text{ if } m < N$$

or

$$abs_e(m) = M'_0||p'_1||\ldots||p'_{m-1} \text{ otherwise.}$$

In this system, M'_0 and the p'_i, for $1 \leq i \leq m-1$ are a *modified* Mailer component and *modified* Client components respectively. The p'_i behave exactly the same as the original (*concrete*) Client components except that they send/receive messages only to/from other concrete components or the abstract component (representing any of the other Client components). The $Mailer'$ component behaves exactly the same as the original $Mailer$ component except that the $Mailer'$ component no longer writes to (the associated channels of) any of the components $p_m, p_{m+1}, \ldots, p_{N-1}$ and a read from such components is replaced by a non-deterministic choice. Thus the Mailer is modified in the same way that the p_i, $0 \leq i \leq m-1$ were modified in the telephone system abstraction.

The component $Abstract_e(m)$ encapsulates part of the observable behaviour of all of the abstracted components. The component $Abstract_e(m)$ has $id = m$ and can send messages to concrete components or to another abstracted component (via the *Mailer* component in each case).

In particular, suppose that $S = M_0||p_1||p_2||\ldots||p_{N-1}$ is a system of email components in which at least the features F are present. Let ϕ be a property and suppose that only components $p_1, p_2, \ldots, p_{m-1}$ are involved in the features F or in ϕ (we will assume that the Mailer component is not involved in the property, for ease of notation). Regardless of whether components $p_m, p_{m+1}, \ldots, p_{N-1}$ have associated features or not (again, with some conditions attached), we will show that if ϕ holds for the model associated with $abs_e(m)$ (namely $\mathcal{M}_{abs_e(m)}$), then it holds for $\mathcal{M}(S)$. This case is illustrated in figure 5.

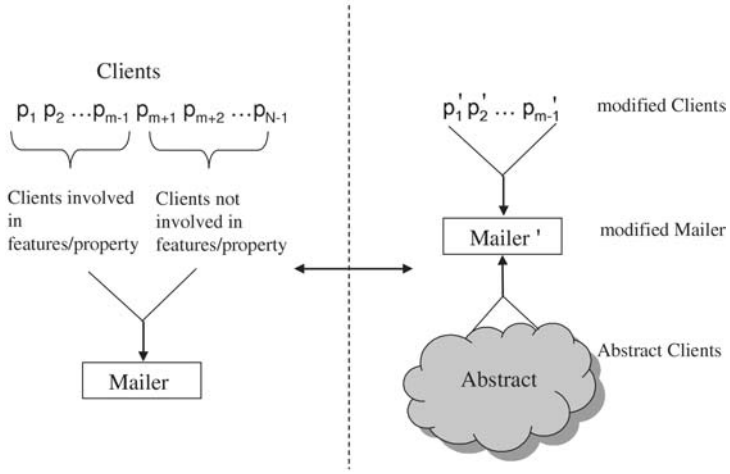


Fig. 5. Abstraction technique for N-Client email model

Because the inline functions relating to all of the features apart from encryption and decryption are called by the Mailer component (so behaviour of the Mailer depends on the features subscribed to by all parties in a communication, including abstract components), our results for the email case only hold when the abstracted components have no features apart from possibly the encryption or decryption features. We can prove the following:

Theorem 2. *Let $S = M_0||p_1||p_2||\ldots||p_{N-1}$ be a system of email components in which at least the features F are present, and ϕ a property.*

1. *If the total index set of $F \cup \{\phi\}$ is $\{1, 2, \ldots, m-1\}$ then if components $p_m, p_{m+1}, \ldots, p_{N-1}$ do not have any features apart from possibly encryption and/or decryption, then $\mathcal{M}_{abs_e(m)} \models \phi$ implies that $\mathcal{M}(S) \models \phi$.*
2. *If the total index set of $F \cup \{\phi\}$ is $\{i_1, i_2, \ldots, i_{m-1}\}$ and σ the permutation that maps j to i_j for $1 \leq j \leq m-1$ and m to m' where m' is the smallest element of $\{1, 2, \ldots, N-1\} \setminus \{i_1, i_2, \ldots, i_{m-1}\}$, then if components $p_{\sigma(m)}, p_{\sigma(m+1)}, \ldots, p_{\sigma(N-1)}$ do not have features apart from possibly encryption and/or decryption, $\mathcal{M}_{abs_e(m)} \models \phi$ implies that $\mathcal{M}(S) \models \sigma(\phi)$.*

The proof is similar to that of Theorem 1, and is omitted here.

9 The Abstract Approach and Feature Interaction Analysis

The correctness of our approach, as presented above, is based upon theorems of the form

$\mathcal{M}_{abs(m)} \models \phi \Rightarrow \mathcal{M}(S) \models \phi$.

In the context of feature interaction detection, this means "if there is no interaction in the network of size m, with the abstract component, then we can infer that there is no interaction for *any* network (of arbitrary size)".

But what can we infer when there *is* an interaction? Namely, what can we infer when $\mathcal{M}_{abs(m)} \not\models \phi$? We cannot necessarily infer $\mathcal{M}(S) \not\models \phi$, because one is not a conservative extension of the other, i.e. there is only a simulation relationship, not a bisimulation relationship. For example, because of increased non-determinism, there may be additional loops in $\mathcal{M}_{abs(m)}$ which are not possible in any instance of $\mathcal{M}(S)$. Thus some liveness properties may actually hold in $\mathcal{M}(S)$, when they do *not* hold in $\mathcal{M}_{abs(m)}$. However, if the property does not hold for a network of size m (*without* the abstract component), i.e. $\mathcal{M}(p_{i_0}||p_{i_1}||\ldots||p_{i_{m-1}}) \not\models \phi$ (or $\mathcal{M}(p_{i_1}||p_{i_2}||\ldots||p_{i_{m-1}}) \not\models \phi$ in the email example) then we can infer $\mathcal{M}(S) \not\models \phi$. In practice, we have yet to encounter a false negative.

In the email example, for all combinations of features and associated property, $m \leq 4$ and full verification is possible. However, for some pairs of features in the telephone example, full analysis requires us to test scenarios where $m = 5$. For example, to fully analyse the pair of features CFU and TCS we must verify that, if $User[j]$ forwards to $User[k]$ and $User[l]$ screens calls from $User[m]$ then the CFU property (see [6]) holds. The CFU property has 3 parameters: j and k (as above) and a further parameter i. Hence, if i, j, k, l and m are all distinct, we have $m = 5$. In some situations where $m = 5$, and in a few (very rare) cases where $m = 4$, full verification is not possible under our current memory restriction of 3Gb.

10 Conclusions

Features are a structuring mechanism for *additional* functionality. When several features are invoked at the same time, for the same, or different components, the features may not interwork. This is known as *feature interaction*. In this paper we take a property based approach to feature interaction detection; this involves checking the validity (or not) of a temporal property against a given system model. We have considered two example systems: a telecommunications system with peer to peer communication, and a client server email system.

A challenge for feature interaction analysis, indeed a challenge for reasoning about any system of components, is to infer properties for networks of *any* size, regardless of features associated with components, and the underlying communication structure. To solve this, for some cases, we have developed an inference mechanism based on abstraction. The key idea is to model-check a system consisting of a constant number (m) of components together with an *abstract* component representing any number of other (possibly featured) components. The approach is sound because there is a simulation between the fixed size system and any system, based on data abstraction. We have applied our approach to both examples and give a upper bound for the value of m (5).

The techniques developed here are motivated by feature interaction analysis, but they are also applicable to reasoning about networks of other types of components such as objects and agents, provided there is a suitable data abstraction and characterisations of the of observable behaviour of sets of components. The

results can also inform testing. For example, the upper bound m allows one to configure (finite) tests which ensure complete coverage.

References

1. D. Amyot and L. Logrippo, editors. *Feature Interactions in Telecommunications and Software Systems VII*. IOS Press (Amsterdam), June 2003.
2. Krzysztof R. Apt and Dexter C. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22:307–309, 1986.
3. L. G. Bouma and H. Velthuijsen, editors. *Feature Interactions in Telecommunications Systems*. IOS Press (Amsterdam), May 1994.
4. M. Calder and E. Magill, editors. *Feature Interactions in Telecommunications and Software Systems VI*. IOS Press (Amsterdam), 2000.
5. M. Calder, E. Magill, and S. Kolberg, Reiff-Marganiec. Feature interaction: A critical review and considered forecast. *Computer Networks*, 41/1:115 – 141, 2003.
6. M. Calder and A. Miller. Using SPIN for feature interaction analysis - a case study. In M.B. Dwyer, editor, *Proceedings of the 8th International SPIN Workshop (SPIN 2001)*, volume 2057 of *Lecture Notes in Computer Science*, pages 143–162, Toronto, Canada, May 2001. Springer-Verlag.
7. Muffy Calder and Alice Miller. Automatic verification of any number of concurrent, communicating processes. In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE 2002)*, pages 227–230, Edinburgh, UK, September 2002. IEEE Computer Society Press.
8. Muffy Calder and Alice Miller. Generalising feature interactions in email. In Amyot and Logrippo [1], pages 187–205.
9. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, 1999.
10. E.M. Clarke, O. Grumberg, and S. Jha. Verifying parameterized networks using abstraction and regular languages. In Insup Lee and Scott A. Smolka, editors, *Proceedings of the 6th International Conference on Concurrency Theory (CONCUR '95)*, volume 962 of *Lecture Notes in Computer Science*, pages 395–407, Philadelphia, PA., August 1995. Springer-Verlag.
11. E.M. Clarke, O. Grumberg, and D Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, January 1994.
12. S.J. Creese and A.W. Roscoe. Data independent induction over structured networks. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'00)*, volume II, Las Vegas, Nevada, USA, June 2000. CSREA Press.
13. P. Dini, R. Boutaba, and L. Logrippo, editors. *Feature Interactions in Telecommunication Networks IV*. IOS Press (Amsterdam), June 1997.
14. E. Allen Emerson and Vineet Kahlon. Reducing model checking of the many to the few. In David A. McAllester, editor, *Automated Deduction - Proceedings of the 17th International Conference on Automated Deduction (CADE 2000)*, volume 1831 of *Lecture Notes in Computer Science*, pages 236–254, Pittsburgh, PA, USA, June 2000. Springer-Verlag.
15. E. Allen Emerson and Kedar S. Namjoshi. Reasoning about rings. In *Conference Record of the 22nd Annual ACM Symposium on Principles of Programming Languages (POPL '95)*, pages 85–94, San Francisco, California, January 1995. ACM Press.

16. Steven M. German and A. Prasad Sistla. Reasoning about systems with many processes. *Journal of the ACM*, 39(3):675–735, July 1992.
17. R.J. Hall. Feature interactions in electronic mail. In Calder and Magill [4], pages 67–82.
18. Gerard J. Holzmann. The model checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
19. *IN Distributed Functional Plane Architecture*, recommmendation q.1204, ITU-T edition, March 1992.
20. C. Norris Ip and David L. Dill. Verifying systems with replicated components in Murϕ. *Formal Methods in System Design*, 14:273–310, 1999.
21. K. Kimbler and L.G. Bouma, editors. *Feature Interactions in Telecommunications and Software Systems V*. IOS Press (Amsterdam), September 1998.
22. M. Kolberg, E. H. Magill, D. Marples, and S. Reiff. Results of the second feature interaction contest. In Calder and Magill [4], pages 311–325.
23. R. P. Kurshan and K.L. McMillan. A structural induction theorem for processes. In *Proceedings of the eighth Annual ACM Symposium on Principles of Distrubuted Computing*, pages 239–247. ACM Press, 1989.
24. Robert P. Kurshan, M. Merritt, A. Orda, and S.R. Sachs. A structural linearization principle for processes. *Formal Methods in System Design*, 5(3):227–244, December 1994.
25. A. Roychoudhury and I. V. Ramakrishnan. Automated inductive verification of parameterized protocols. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Proceedings of the thirteenth International Conference on Computer-aided Verification (CAV 2001)*, volume 2102 of *Lecture Notes in Computer Science*, pages 25–37, Paris, France, July 2001. Springer-Verlag.
26. Pierre Wolper and Vinciane Lovinfosse. Properties of large sets of processes with network invariants (extended abstract). In J. Sifakis, editor, *Proceedings of the International Workshop in Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 68–80, Grenoble, France, June 1989. Springer-Verlag.

Software Evolution through Dynamic Adaptation of Its OO Design

Walter Cazzola[1,⋆], Ahmed Ghoneim[2], and Gunter Saake[2]

[1] Department of Informatics and Communication,
Università degli Studi di Milano, Italy
cazzola@dico.unimi.it
[2] Institute für Technische und Betriebliche Informationssysteme,
Otto-von-Guericke-Universität Magdeburg, Germany
{ghoneim,saake}@iti.cs.uni-magdeburg.de

Abstract. In this paper we present a proposal for safely evolving a software system against run-time changes. This proposal is based on a reflective architecture which provides objects with the ability of dynamically changing their behavior by using their design information. The meta-level system of the proposed architecture supervises the evolution of the software system to be adapted that runs as the base-level system of the reflective architecture. The meta-level system is composed of cooperating components; these components carry out the evolution against sudden and unexpected environmental changes on a reification of the design information (e.g., object models, scenarios and statecharts) of the system to be adapted. The evolution takes place in two steps: first a meta-object, called *evolutionary meta-object*, plans a possible evolution against the detected event then another meta-object, called *consistency checker meta-object* validates the feasibility of the proposed plan before really evolving the system. Meta-objects use the system design information to govern the evolution of the base-level system. Moreover, we show our architecture at work on a case study.

Keywords: Software Evolution, Reflection, Consistency Validation, Dynamic Reconfiguration, UML, XMI.

1 Introduction

In advanced object-oriented information systems related to engineering applications, classes and, therefore, their instances are subjected to frequent adaptations during their life cycle. This applies to the structure of class definitions and the behavior of their objects. In the last decade, several object-oriented systems were designed to address the problem of adapting object structure and behavior to meet new application requirements (see for example [9, 14]).

Nowadays a topical issue in the software engineering research area consists of producing software systems able to adapt themselves to environmental changes by adding new and/or modifying existing functionalities. *Computational reflection* [4,15] provides one of the most used mechanisms for getting software adaptability.

⋆ Walter Cazzola's work has been partially supported by Italian MIUR (FIRB "Web-Minds" project N. RBNE01WEJT_005).

M.D. Ryan et al. (Eds.): Objects, Agents, and Features, LNCS 2975, pp. 67–80, 2004.

A software system with a long life span, must be able to dynamically adapt itself to face unexpected changes in its environment avoiding a long out-of-service period for maintenance. The evolution of the design of a software system is determined by the evolution of the behavior of its components and of the interactions among them. Several design elements govern such aspects: *class/object diagrams statecharts* and *sequence diagrams*. Run-time system evolution also involves such aspects, therefore, design information should be used to concert run-time evolution as well. Design information provide the right mechanism to grant the consistency of the evolved system against the system requirements.

A reflective architecture represents the perfect structure that allows running systems to adapt themselves to unexpected external events, i.e., to consistently evolve. In [6] we described a reflective architecture for system evolution at run-time. In such a framework, the system running in the base-level is the one prone to be adapted, whereas software evolution is the nonfunctional feature realized by the meta-level system. Evolution takes place exploiting design information concerning the running systems.

To correctly evolve[1] the base-level system, the meta-level system must face many problems. The most important are: (1) to determine which events cause the need for evolving the base-level system (2) how to react on events and the related evolutionary actions (3) how to validate the consistency and the stability of the evolved system and eventually how to undo the evolution, (4) to determine which information allows system evolution and/or is involved in the evolution.

In [5], we introduced a pattern language modeling the general behavior of the meta-level components and their interactions during the evolutionary process.

The rest of the paper is organized as follows: section 2 provides a brief overview of the tools we have adopted in our work; section 3 describes our reflective architecture (the evolutionary mechanism) and the evolutionary engine related with the architecture and its rules; section 4 describes the application of our reflective approach to software evolution by an example. Finally, in sections 5 and 6 we survey some related work, draw our conclusions and present some future work.

2 Background

2.1 Computational Reflection

Computational reflection or *reflection* for short is the ability of a system to watch its own computation and possibly change the way it performs. Observation and modification imply an "underlay" that will be observed and modified. Since the system reasons about itself, the "underlay" modifies itself, i.e. the system has a *self-representation* [15].

A *reflective architecture* logically models a system in two layers, called *base-level* and *meta-level*[2]. The base-level realizes the *functional* aspect of the system, whereas

[1] By the sentence *correctly evolve a system* we mean the fact that evolution takes place only when the system remains consistent and stable after evolution.

[2] In the sequel, for simplicity, we refer to the "part of the system working in the base-level or in the meta-level" respectively as base-level and meta-level.

the meta-level realizes the *nonfunctional* aspect of the system. Functional and nonfunctional aspects discriminate among features, respectively, *essential or not* for committing with the given system requirements. Security, fault tolerance, and evolution are examples of nonfunctional requirements[3]. The meta-level is *causally connected* to the base-level, i.e., the meta-level has some data structures, generally called *reification*, representing every characteristic (structure, behavior, interaction, and so on) of the base-level. The base-level is continuously kept consistent with its reification, i.e., each action performed in the base-level is *reified* by the reification and vice versa each change performed by the meta-level on the base-level reification is *reflected* on the base-level. More about the reflective terminology can be learned from [4, 15].

Reflection is a technique that allows a system for maintaining information about itself (meta-information) and using this information to change (adapt) its behavior. This is realized through the casual connection between the base- (the monitored system) and the meta-level (the monitoring system).

2.2 Design Information

Our approach to evolution uses *design information* as a knowledge base for getting system evolution. Design information consists of data related to the design of the system we want to evolve. UML is the adopted formalism for representing design information.

The *unified modeling language* (UML) [3, 11] has been widely accepted as the standard object-oriented modeling language for modeling various aspects of software systems. UML is an extensible language, it provides mechanisms (stereotypes, and constraints) that allow introducing new elements for specific domains if necessary, such as web applications, database applications, business modeling, software development processes, and so on. A *stereotype* provides an extensibility mechanism for creating new kinds of model elements derived from existing ones, whereas *constraints* can be used to refine the behavior of each model element.

The design information we consider are related to two categories: system structure and behavior. *Structural design information* is an explicit description of the structure of the base-level objects. This includes the number of attributes and their data type. *Behavioral design information* describes the computations and the communications carried out by the base-level objects. It includes objects behavior, collaboration between objects, and the state of the objects. Structure and behavior of the system are modeled by class diagrams, sequence diagrams and state diagrams.

3 Software Evolution through Reflection

The main goal of our approach consists of evolving a software system to face environmental and requirement changes and validate the consistency of such an evolution. This goal is achieved by:

[3] The borderline between what is a functional feature and what is a nonfunctional feature is quite confused because it is tightly coupled to the problem requirements. For example, in a traffic control system the security aspect can be considered nonfunctional whereas security is a functional aspect of an auditing system.

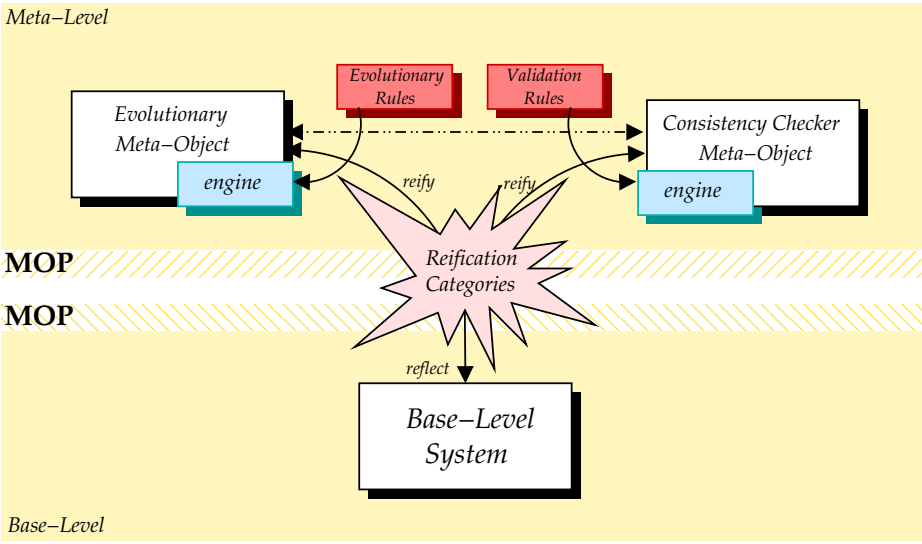


Fig. 1. Reflective architecture designed for the evolution of software systems.

- adopting a reflective architecture which dynamically drives the evolution of the software system through its design information when an event occurs; this has been made possible by moving design information from design- to run-time.
- using two sets of rules which respectively describes how evolution takes place and when the system is consistent; these rules are used by the decisional components of the reflective architecture but not by the system that must be evolved;
- replanning the design information of the system and reflecting the changes on the running system.

In the rest of this section we give a brief overview of the reflective architecture and its components, we show how these components work and the manipulation of the design information.

3.1 The Reflective Architecture

To render a system self-adapting[4], we encapsulate it in a two-layers reflective architecture as shown in Fig. 1. The base-level is the system that we want to render self-adapting whereas the meta-level is a second software system which reifies the base-level design information and plans its evolution when particular events occur. Reflective properties as transparency [20, 21] and separation of concerns [13] provide the meta-level with the mechanism for carrying out the evolution of the base-level code and behavior without having previously foreseen such an adaptation for the system.

[4] By the sentence *to render a system self-adapting* we mean that such a system is able to change its behavior and structure in accordance with external events by itself.

At the moment, we just take in consideration two kinds of software adaptation: *structural* and *behavioral* evolution. This limitation is due to the fact that, at the moment, we only reify the following design information related to the base-level:

- *object and class diagrams*, which describes classes, objects and their relations; this model represents the structural part of the system;
- *sequence diagrams*, which trace system operations between objects (inter-object connection) for each use case at a time; and
- *statecharts*, which represent the evolution of the state of each object (intra-object connection) in the system.

The approach can be easily extended to observe and manipulate all the other diagrams provided by UML such as *use case*, *activity diagrams*.

The meta-level is responsible of dynamically adapting the base-level and it is composed of some special meta-objects, called *evolutionary meta-objects*. There are two types of evolutionary meta-objects: the *evolutionary* and the *consistency checker* meta-objects. Their goals consists of consistently evolving the base-level system. The former is directly responsible for planning the evolution of the base-level through adding, changing or removing objects, methods, and relations. The latter is directly responsible for checking the consistency of the planned evolution and of really carrying out the evolution through the causal connection relation.

The base-level system and its design information is reified in *reification categories* in the meta-level (see section 3.3 for more details). Classic reflection takes care of reifying the state and every other dynamic aspect of the base-level system, whereas the design information provides a reification of the design aspects of the base-level system such as its architecture and the collaborations among its components. The reification categories content is the main difference of our architecture with respect to standard reflective architectures. Usually, reifications represent the base-level system behavior and structure not its design information. Reification categories can be considered as representatives of the base-level system design information in the meta-level. Both evolutionary and consistency checker meta-objects directly work on such representatives and not on the real system, this allows a safe approach to evolution postponing every change after validation checks. As described in [5] when an external events occur, the evolutionary meta-object proposes an evolution as a reaction to the consistency checker meta-object which validates the proposal and schedules the adaptation of the base-level system if the proposal is accepted.

3.2 Decisional Engines and Evolutionary Rule Sets

Adaptation and validation are respectively driven by a set of rules which define how to adapt the system in accordance with the detected event and the meaning of system consistency.

To give more flexibility to the approach, these rules are not hardwired in the corresponding meta-object rather they are passed to a sub-component of the meta-objects themselves, respectively called *evolutionary* and *validation engines*, which interpret them. Therefore, each meta-object has two main components: (i) the core which interacts with the rest of the system (e.g., detecting external events/adaptation proposals, or manipulating the reification categories/applying the adaptation on the base-level

system) and implementing the meta-object's basic behavior, and (ii) the engine which interprets the rules driving the meta-object's decisions.

In this paper, for sake of simplicity, we express both evolutionary and validation rules by using the formalism for *event-condition-action (ECA)* [1, 8] rules. Rules are usually written in the following form:

on *event* **if** *conditions* **do** *actions*

where *event* represents the event which should ignite the evolution of the base-level system, *conditions*, and *actions*, respectively, represent the conditions the engine must validate when the *event* occurs and the actions the engine must carry out for adapting the system against the occurred *event*. Both *events* and *conditions* involve the base-level reification (see section 3.3 for details on reifications). The engines (both evolutionary and consistency checker) interpreting these rules are simply state machines indexed on events and conditions.

Both rules and engines working on them are tightly bound but completely unbound from the rest of the reflective architecture. Therefore, to adapt our approach to use rules specified with a different formalism is quite simple; we have just to substitute the engine with one able to interpret the chosen formalism. Of course, the engines must be able to interact with the rest of the architecture as described in the following algorithm. More complex and powerful approaches are under development.
In general, adaptation takes place as follows:

- the meta-level reifies the base-level design information and the system itself into reification categories;
- the evolutionary meta-object waits for an event that needs the adaptation of the base-level system; when such an event occurs it starts to plan evolution:
  - through the design information of the base-level system, it detects which base-level components might be involved in the evolution; then
  - it informs its engine about the occurred event and components involved in the evolution;
  - the evolutionary engine decides which evolutionary rule (or which group of evolutionary rules) is better to apply; then
  - it designs the evolutionary plan by applying the chosen evolutionary rule (or group of rules);
- the evolutionary meta-object passes the evolutionary plan to the consistency checker meta-object which must validate the proposed evolutionary plan before rendering the adaptation effective:
  - the consistency checker meta-object demands the validation phase to the validation engine;
  - the validation engine validates the proposed evolutionary plan by using its validation rules and the base-level system design information.
- if the proposed evolutionary plan is considered sound the consistency checker meta-object schedules the base-level system adaptation in accordance with such an evolutionary plan; otherwise the consistency checker meta-object returns an error message to the evolutionary meta-objects restarting the adaptation phase.

The evolutionary plan proposed by the evolutionary meta-object is a manipulation of the design information of the base-level system. The causal connection is responsible of modifying the model of the base-level system accordance with the proposed evolution. The adopted mechanism for transposing design information in the real system is based on the UML virtual machine [17].

The most important side-effect of this approach is represented by the fact that adaptation can take place also on nonstoppable systems because it does not require that the base-level system stops during adaptation, but it only needs to define when it is safe to carry out the adaptation.

3.3 Reification and Reflection by Using Design Information

We have talked about reifying and reflecting on design information of the base-level system whereas such design information simply feed the meta-level system during system bootstrap and drive its meta-computations during the evolution of the base-level system.

When an event occurs, the design information related to the base-level entities, that can be involved by the event, are used by the evolutionary and the consistency checker meta-objects for driving the evolution of such base-level entities (as described in the previous algorithm).

Design information identifies which entities are involved by the event (object/class and state diagrams), their behavior (sequence diagrams) and how the event can be propagated in the base-level system (collaboration diagrams). Therefore introspection and intercession on large systems become simpler than using standard reflective approaches because the design information provide a sort of index on the base-level entities and their interactions.

Moreover design information is the right complement to the base-level system reification built by the standard causal connection. Meta-objects consult and manipulate the design information in order to get information that otherwise are not easily accessible from the running system, e.g., the collaboration among objects. Design information is also used as a testbed for manipulation because they give a easily accessible overview of global features as inter-objects collaborations.

UML specifications provide graphical data that usually exist at design-time and are difficult to manage at run-time. Whereas, our meta-objects require such a specifications at run-time for driving the evolution. We chose to overcome this problem by encoding the design information in XML, in particular with the XMI standard [16]. XMI provides a translation of UML diagrams in a text-based form more suitable for run-time manipulation.

The XMI standard gives a guideline for translating each UML diagram in XML. Each diagram is assimilated to a graph whose nodes are the diagram's components (e.g., classes, states and so on), and arcs represents the relation among the components. The graph is decorated with XML tag describing the properties of the corresponding UML component. In our architecture, we use the XMI code generated by Poseidon4UML [2]. Poseidon4UML provides us with a tool for drawing UML diagrams and for generating the corresponding XMI code. Figure 2 shows a simple class diagram,

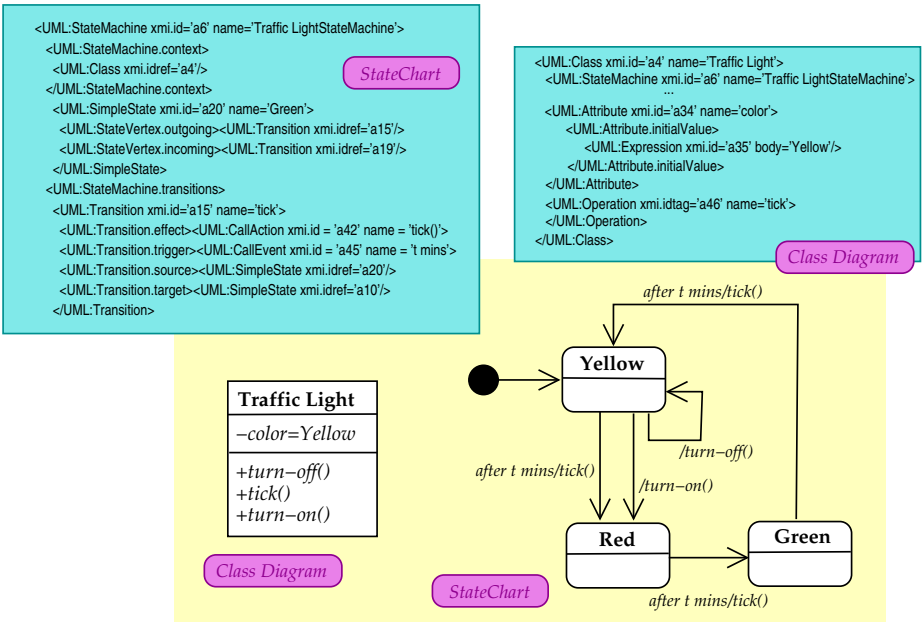


Fig. 2. Poseidon4UML's XMI translation of a simply class diagram and the related statechart.

a possible statechart for that and the corresponding XMI code (shortened for the sake of space) generated by Poseidon4UML.

At system bootstrap, the meta-level reifies the design information of the base-level, that is, the meta-level loads into the reification categories (each categories is devoted to an aspect of the base-level) the XMI representation of design information of the base-level. In this way we render accessible UML data-model to the meta-objects.

The XMI schemas are tightly linked with the base-level components. The evolutionary meta-object modifies these schemas introducing some specific XMI elements, providing the consistency checker with the necessary pieces of information for validating the evolutionary plan and for effectively modifying the base-level. Some of these elements are:

- `XMI.difference` is the element used to describe differences from the initial model;
- `XMI.delete` represents a deletion from the base model;
- `XMI.add` represents an addition to the base model; and
- `XMI.replace` represents a replacement of a model construct with another model construct in the initial model.

As described in [12], we can create new UML models from XMI schemas, therefore the evolutionary plan, which is a group of modified XMI schemas, can be reverted into UML diagrams. Basically, this reciprocity between UML diagrams and XMI schemas allows us maintaining the causal connection between base- and meta-level.

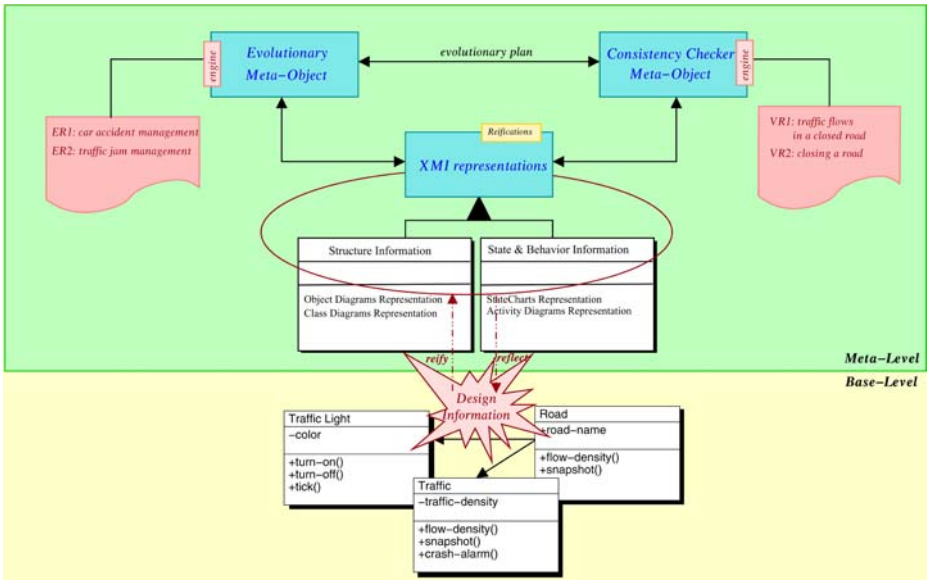


Fig. 3. XMI reification data and UML UTCS flow density.

4 Urban Traffic Control System: A Case Study

When designing *urban traffic control systems* (UTCS), the software engineer will face many issues such as distribution, complexity of configuration, and reactivity to the environment evolution. Moreover, modern cities have to face a lot of unexpected hard to plan problems such as traffic lights disruptions, car crashes, traffic jams and so on. In [18] these issues and many others are illustrated.

The evolution of complex urban agglomerates have posed significant challenges to city planners in terms of optimizing traffic flows in a normally congested traffic network. Simulation and analysis of such systems require modeling the behavioral, structural and physical characteristics of the road system. This model includes mobile entities (e.g., cars, pedestrians, vehicular flow, and so on) and fixed entities (e.g., roads, railways, level crossing, traffic lights and so on).

Of course, the UTCS, due to its complexity, cannot be considered as a whole case study. In this section, we describe our approach to evolution involving just three components of the UTCS: *road*, *traffic light* and *traffic*. Figure 3 shows how this fragment of the UTCS is integrated with our reflective approach to evolution, and how the UTCS design information is managed by using the XMI encoding.

In this example we consider as unexpected events only car accidents. Therefore, we will show some evolutionary and validation rules that, used in conjunction with our approach, force the UTCS evolution for dealing with problems due to car accidents, e.g., traffic jams.

4.1 Evolutionary and Validation Rules against Car Accidents

For UTCS to deal with traffic jams must monitor vehicular flow density[5], and the status of every involved entity (both fixed and mobile). The evolutionary meta-object detects when the vehicular flow augments in a specific street and plans to adapt the current traffic schedule (i.e., the UTCS behavior) to face the problem.

Evolutionary Rules

ER$_1$ (car accident management): when there is a notification of a car accident in a specific road and the traffic in that road is stuck then such a street must be closed and the traffic flow hijacked towards the adjacent streets.

> **on** *(a car accident in street R_i has been detected)*
> **if** *(flow density of R_i is too elevate)*
> **do** *closes R_i and reschedules the traffic lights so that cars avoid to pass through R_i.*

Obviously, the traffic lights rescheduling implies a change in the statechart of the traffic lights, whereas closing the road means to add a new traffic light in the system. The evolutionary meta-object will render effective these considerations manipulating the corresponding XML schemas and forming the evolutionary plan to pass to the consistency checker meta-object.

ER$_2$ (traffic jam management): the density of the vehicular flow is constantly monitored road by road thanks to automatic cameras installed at the entrance of each road. These cameras take a snapshot of the traffic entering in the road every few seconds (delta that can be changed to have a more timely reaction), consecutive snaps are compared to establish if the vehicular flow has overcome a given tolerance threshold, if so the temporization of the traffic lights in the corresponding area are modified to allow a more fluid circulation in the road.

> **on** *(comparing two consecutive snapshots)*
> **if** *(the traffic flow has increased overcoming the tolerance threshold)*
> **do** *control the traffic lights and modify their temporization.*

As for **ER$_1$**, changing the traffic light temporization implies a modification of the statechart associate to the involved traffic light.

Validation Rules

VR$_1$ (traffic flows in a closed road): when the evolutionary meta-object proposes an evolutionary plan in which traffic is inhibited to a certain road, the consistency checker must verify that there is not any road whose traffic flows in the inhibited road.

[5] UTCS is supported by CCD-Cameras and movement sensors installed in every important nexus [18]. CCD-cameras take a photo every second and by comparing these photos, we can estimate the traffic flowing density. Sensors will notify anomaly events that cannot be detected by CCD-cameras like traffic light disruptions or damages to the road structure.

on *(traffic has been inhibited to road R_i)*
if *(there is a road R_j whose traffic still flows into R_i)*
do *inconsistency has been detected, reject the plan.*

The checking for consistency implies a complete scan of the object models and statecharts of the roads whose traffic usually flows in the closed road.

Note that we have decided of rejecting the evolutionary plan but another strategy should consider to fix the evolutionary plan against the detected problems instead of rejecting it.

VR_2 (closing a road): another important aspect that must be validated before rendering effective the planned evolution is to determine when it is safe to schedule the changes. In our example this mean to wait that all cars are halted at the entrance of the road to close before changing the direction of the vehicular flow.

on *(evolutionary plan has been authorized) $\wedge$ (road R_i must be closed)*
if *(no car is entering in R_i)*
do *turns red the traffic lights in R_i and applies the evolutionary plan.*

This rule monitors the traffic light statecharts and intervenes on that when it is feasible before applying the evolutionary plan.

The rules showed in this section do not pretend to cover every aspect of the evolution but they want only to give a glance at the possibilities offered by our approach. Moreover the rules are expressed by using the natural language because the scope of this paper consists of describing our approach to evolution and we prefer to avoid complicated formalisms that would have obscured the simplicity of the mechanism.

5 Related Work

Several other researchers have proposed a mechanism for dynamic evolution by using a reflective architecture and design information. The system we consider in this short overview are UML virtual machine [17], The K-Component Architecture [10], Architectural Reflection [7] and design enforcement [19].

In [17] has been presented the architecture for a UML virtual machine. The virtual machine has a logical architecture that is based on the UML four-level modeling architecture and a physical architecture that realizes the logical architecture as an object-oriented framework.

Dowling and Cahill [10] have proposed a meta-model framework named *K-components*, that realizes a dynamic, self-adaptive architecture. It reifies the features of the system architecture, e.g., configuration graph, components and connectors. This model presents a mechanism for integrating the adaptation code in the system, and a way to deal with system integrity and consistency during dynamic reconfiguration.

Cazzola et al. [7] have presented a novel approach to reflection called *architectural reflection* which allows dynamic adaptation of a system through its design information. This has been possible moving the system software architecture from design-time to run-time. Software architecture manipulation allows adaptation in-the-large of the sys-

tem, i.e., we can add and remove components but we cannot add functionalities to a component.

In [19] has been presented a method for design enforcement, based on a combination of reflection and state machine diagrams. Combining concepts of concurrent object-oriented design, finite state diagrams, and reflection leads to increase the reliability of the systems, by insuring that objects work in accordance with their design.

6 Conclusions and Future Work

The main topic of our work concerns with software adaptability. In this paper we have presented: i) a reflective architecture for dynamically and safely evolving a software system; and ii) the decisional engines and their rules which govern such an evolution. Finally, we have shown on a case study how to instruct our reflective architecture to adapt itself to unexpected events and how the evolution takes effect.

Our approach to software evolution has the following benefits:

- evolution is not tailored to a specific software system but depends on its design information;
- evolution is managed as a nonfunctional features, therefore, can be added to every kind of software system without modifying it; and
- evolution strategy is not hardcoded in the system but it can be dynamically changed by substituting the evolutionary and validation rules.

Unfortunately there are also some drawbacks: (i) we need a mechanism for converting UML diagrams in the corresponding XMI schemas (problem partially overcome by using *Poseidon4UML* [2]); (ii) decomposing the evolution process in evolution and consistency validation could be inadequate for evolving systems with tight time constraints.

In future work, we plan to overcome the cited drawbacks and to implement a prototype of the described architecture by using OpenJava for supporting the causal connection among meta-level representation and the base-level system and a scripting language such as Ruby or Python for specifying and interpreting the rules.

References

1. James Bailey, Alexandra Poulovassilis, and Peter T. Wood. An event-condition-action language for XML. In *Proceedings of the 11th International World Wide Web Conference, WWW2002*, pages 486–495, Honolulu, Hawaii, USA, May 2002. ACM Press.
2. Marko Boger, Thorsten Sturm, and Erich Schildhauer. *Poseidon for UML Users Guide*. Gentleware AG, Vogt-Kölln-Str. 30, D-22527 Hamburg, Germany, 2000.
3. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison-Wesley, Reading, Massachusetts, third edition, February 1999.
4. Walter Cazzola. Evaluation of Object-Oriented Reflective Models. In *Proceedings of ECOOP Workshop on Reflective Object-Oriented Programming and Systems (EWROOPS'98)*, in 12th European Conference on Object-Oriented Programming (ECOOP'98), Brussels, Belgium, on 20th-24th July 1998. Extended Abstract also published on ECOOP'98 Workshop Readers, S. Demeyer and J. Bosch editors, LNCS 1543, ISBN 3-540-65460-7 pages 386-387.

5. Walter Cazzola, James O. Coplien, Ahmed Ghoneim, and Gunter Saake. Framework Patterns for the Evolution of Nonstoppable Software Systems. In Pavel Hruby and Kristian Elof Søresen, editors, *Proceedings of the 1st Nordic Conference on Pattern Languages of Programs (VikingPLoP'02)*, pages 35–54, Højstrupgård, Helsingør, Denmark, on 20th-22nd of September 2002. Microsoft Business Solutions.
6. Walter Cazzola, Ahmed Ghoneim, and Gunter Saake. Reflective Analysis and Design for Adapting Object Run-time Behavior. In Zohra Bellahsène, Dilip Patel, and Colette Rolland, editors, *Proceedings of the 8th International Conference on Object-Oriented Information Systems (OOIS'02)*, Lecture Notes in Computer Science 2425, pages 242–254, Montpellier, France, on 2nd-5th of September 2002. Springer-Verlag. ISBN: 3-540-44087-9.
7. Walter Cazzola, Andrea Savigni, Andrea Sosio, and Francesco Tisato. Rule-Based Strategic Reflection: Observing and Modifying Behaviour at the Architectural Level. In *Proceedings of 14th IEEE International Conference on Automated Software Engineering (ASE'99)*, pages 263–266, Cocoa Beach, Florida, USA, on 12th-15th October 1999.
8. Stefan Conrad and Can Türker. Prototyping Object Specifications Using Active Database Systems. In A. Emre Harmancı, Erol Gelenbe, and Bulent Örencik, editors, *Proceedings of the 10th International Symposium on Computer and Information Sciences (ISCIS X)*, Volume I, pages 217–224, Kuşadası, Turkey, October 1995.
9. Jim Dowling and Vinny Cahill. Building a Dynamically Reconfigurable Minimum CORBA Platform with Components, Connectors and Language-Level Support. In *Proceedings of the IFIP/ACM Middleware 2000 Workshop on Reflective Middleware*, New York, NY, USA, April 2000. Springer-Verlag.
10. Jim Dowling and Vinny Cahill. The K-Component Architecture Meta-Model for Self-Adaptive Software. In Akinori Yonezawa and Satoshi Matsuoka, editors, *Proceedings of 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection'2001)*, LNCS 2192, pages 81–88, Kyoto, Japan, September 2001. Springer-Verlag.
11. Martin Fowler and Kendall Scott. *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley, Reading, Massachusetts, 1997.
12. Timothy J. Grose, Gary C. Doney, and Brodsky Stephan A. *Mastering XMl: Java Programming with XMl, XML, and UML*. John Willy & Sons, Inc., April 2002.
13. Walter Hürsch and Cristina Videira Lopes. Separation of Concerns. Technical Report NU-CCS-95-03, Northeastern University, Boston, February 1995.
14. Jeff Kramer and Jeff Magee. Analysing Dynamic Change in Distributed Software Architectures. *IEEE Proceedings Software*, 145(5):146–154, October 1998.
15. Pattie Maes. Concepts and Experiments in Computational Reflection. In Norman K. Meyrowitz, editor, *Proceedings of the 2nd Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, volume 22 of *Sigplan Notices*, pages 147–156, Orlando, Florida, USA, October 1987. ACM.
16. OMG. OMG-XML Metadata Interchange (XMI) Specification, v1.2. OMG Modeling and Metadata Specifications available at http://www.omg.org, January 2002.
17. Dirk Riehle, Steven Fraleigh, Dirk Bucka-Lassen, and Nosa Omorogbe. The Architecture of a UML Virtual Machine. In Linda Northrop and John Vlissides, editors, *Proceedings of the 2001 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'01)*, pages 327–341, Tampa Bay, Florida, USA, October 2001. ACM Press.
18. Andrea Savigni, Filippo Cunsolo, Daniela Micucci, and Francesco Tisato. ESCORT: Towards Integration in Intersection Control. In *Proceedings of Rome Jubilee 2000 Conference (Workshop on the International Foundation for Production Research (IFPR) on Management of Industrial Logistic Systems – 8^{th} Meeting of the Euro Working Group Transportation - EWGT)*, Roma, Italy, September 2000.

19. Shaul Simhi, Vered Gafni, and Amiram Yehudai. Combining Reflection and Finite State Diagrams for Design Enforcement. 2(4):269–281, 1997.
20. Robert J. Stroud. Transparency and Reflection in Distributed Systems. *ACM Operating System Review*, 22:99–103, April 1992.
21. Robert J. Stroud and Zhixue Wu. Using Metaobject Protocols to Satisfy Non-Functional Requirements. In Chris Zimmerman, editor, *Advances in Object-Oriented Metalevel Architectures and Reflection*, chapter 3, pages 31–52. CRC Press, Inc., 2000 Corporate Blvd.,N.W., Boca Raton, Florida 33431, 1996.

Modelling and Analysis of Agents' Goal-Driven Behavior Using Graph Transformation*

Ralph Depke and Reiko Heckel

Faculty of Computer Science, Electrical Engineering and Mathematics
University of Paderborn
D-33098 Paderborn, Germany
{depke,reiko}@upb.de

Abstract. Extending mainstream object-oriented concepts, the agent paradigm promotes the concept of *goals* realized by means of *strategies*. To account for such specific aspects, dedicated techniques for *agent-oriented modelling* are required which go beyond standard techniques of object-oriented modelling.
In this paper, an agent-oriented modelling notation for goals and strategies is proposed. Based on graph transformation as semantic domain we show how the behavior of agents can be described in terms of goals and the strategies or communication protocols for achieving them. Model checking is used to verify, in a given situation, that a chosen strategy actually achieves its goal.

1 Introduction

The concepts and technologies of agent-based systems become increasingly attractive to the software industry [11]. In order to benefit from advantages like increased scalability or flexibility of agent architectures, agents substitute more traditional software, or they are integrated into legacy systems. Thus, agent-oriented software development is about to become one aspect of the "normal" software development process.

Today, most software systems are implemented in object-oriented programming languages like C++ or Java, and the analysis and design of such systems is based on object-oriented modelling languages like the UML [12]. Thus, in order to incorporate agent concepts into mainstream software development, an integrated modelling approach for object- and agent-based systems is required.

Agent-oriented modelling can be seen as an extension and specialization of object-oriented modelling to the particular aspects of agents and multi-agent systems. As modelling concepts, agents and objects have complementary roles: Agents act autonomously, driven by their goals and strategies, thereby sensing and reacting to their environment and cooperating with other agents. Objects encapsulate data structures and operations and provide services to other objects.

* Work supported in part by the EC's Human Potential Programme under contract HPRN-CT-2002-00275, [Research Training Network SegraVis].

M.D. Ryan et al. (Eds.): Objects, Agents, and Features, LNCS 2975, pp. 81–97, 2004.

In this sense, Jennings, et al. [15] state that *"There is a fundamental mismatch between the concepts used by object-oriented developers . . . and the agent-oriented view."* However, the view of objects as mere service providers has its origins in the paradigms of sequential OO programming, and is no longer adequate when considering concurrent languages like Java where objects may have their own thread of control. As a modelling abstraction for concurrent objects, the concept of *active object* has been established [12] which has much similarity with the agent paradigm. What is missing, even in active objects, is the idea of goal-driven behavior.

Wooldridge characterizes goal-orientation in programming in the following way (cf. [8], p. 32): "It is not hard to build a system that exhibits goal directed behavior – we do it every time we write a procedure in PASCAL, a function in C, or a method in JAVA. When we write such a procedure, we describe it in terms of the assumptions on which it relies (formally, its pre-condition) and the effect it has if the assumptions are valid (its post-condition). The effects of the procedure are its goal: what the author of the software intends the procedure to achieve."

From a software engineering perspective, a goal corresponds to a functional requirement to be satisfied by the behavior exposed by an object or component. Traditionally, such requirements are given separately from the software in a requirements specification. Goal-*driven* behavior means that goals are used at run-time to *control* the behavior of agents, that is:

- In a *given situation* matching the pre-condition of a goal, its post-condition defines the *desired situation* that the agent tries to establish through its actions. A strategy associated with the goal provides a *decomposition into subgoals*, which can be realized more directly.
- The subgoals are instantiated and tested, in turn, for satisfaction of their pre-conditions. The decomposition is continued hierarchically down to the level of basic operations that are immediately executable.

This conceptual execution model is not unlike the operational semantics of logic programs, which rely on depth-first search and backtracking to select a suitable "strategy" (i.e., clause) for reaching a goal. Since agents are embedded in the real world, which is directly influenced by its actions, backtracking is not always possible. Therefore, it is important to verify statically that strategies achieve their associated goals.

The paper's contribution to this problem is two-fold.

1. We present an implementation-independent representation of goals and strategies, that allows for recursive decomposition, provides an operational interpretation, and is rich enough to capture the reactivity and dynamic nature of agent-based systems.
2. We clarify, what it means for a strategy to achieve its goal and show, how this property can be verified.

We propose the use of graph transformation rules for modelling goals and strategies in a UML-like notation [7, 5]. A graph transformation rule provides

a visual representation of the pre- and post-condition of an operation by means of graphical patterns for the states before and after the execution. The rules come with an operational semantics of *rule application* where, given a graph representing the current state that matches the pre-condition of the rule, this graph is transformed by replacing the occurrence of the pre-pattern with an instance of the post-pattern.

Modelling goals as graph transformation rules provides us with the required operational interpretation for the execution of goals. Moreover, graph transformation systems provide a concurrent model of computation, not unlike that of Petri nets, where sequences of transformation steps are abstracted to partial orders, identifying all sequences which differ only for the ordering of independent steps (see [2]). Thus, independent goals can be established in any order or in parallel, and we do not distinguish the different schedulings in the concurrent semantics.

To capture a notion of correctness of a strategy for reaching a goal, *winning strategies* are introduced. Whether a strategy is a winning strategy, establishing its goal under all circumstances, cannot be verified in general. However, we may use model checking techniques for graph transformation systems [14] to detect errors, thus increasing our confidence in a strategy in the same sense software testing is used to increase confidence in the correctness of some software component.

In the following section we demonstrate how agent-based systems can be modelled using graph transformation. Then, in Section 3 we explain how graph transformation systems can be translated into PROMELA, the input language of the SPIN model checker. Based on this translation we analyze the question, whether a goal is always achieved by a given strategy.

2 Modelling Goals and Strategies of Agent-Based Systems

Systems composed of multiple agents are characterized by goals that are shared between several cooperating agents. Strategies for achieving these kinds of goals have to integrate local subgoals of different agents, interconnected by communication. The most important class of such communication-oriented strategies are *interaction protocols*, see e.g., [10]. Generally, protocols are generic descriptions of messages exchanged between agents and associated state changes. By executing a protocol the subgoals attributed to the participants of the protocol are attained and thus the overall goal of the protocol is reached.

To abstract from the concrete types of agents, participants of a generic protocol are characterized as roles which comprise only those features needed to perform their respective task in the protocol. Thus, agents, roles and protocols are the fundamental notions in our agent-based modelling language AML [6, 5].

We explain by means of a running example the concepts and the syntax of the language. First, we present a protocol in terms of its goal rule and its "implementation" given by a set of rules describing the subgoals. Then, we show

how a lower-level protocol is used in the context of a superior protocol, which instantiates the former in a way similar to a procedure call. Finally, we present a formal semantics for goals and strategies.

2.1 Protocols for Agent Interaction

Protocols are communication strategies in which agents exchange messages in order to reach a common goal. A prominent example is the *English auction protocol* [10]. In this protocol one agent, the auctioneer, communicates with several other agents, the customers, in order to determine a customer with a maximal bid.

The auctioneer tries to maximize the result by repeatedly asking for an increased bid which the customers may or may not accept. This repeated question-and-answer pattern can be realized by another agent interaction protocol, the so-called *contract net protocol*, used by the English auction protocol. We will demonstrate the basic notions of the AML using a model of the contract net protocol.

The contract net protocol is a typical interaction pattern for multi-cast communication in agent-based systems [13]. In the contract net protocol two kinds of roles are involved, the selector role type and the contractor role type. The selector role solicits proposals from the contractor roles by issuing a call for proposals. Contractor roles receiving the call for proposals are able to generate proposals in reply. Once the selector role receives replies from the contractor roles it chooses the one that replied first. The contractor role of the selected proposal will be sent an acceptance message.

An agent interaction protocol is described by a structure diagram and a set of operations which are applied within the protocol. A structure diagram for the contract net protocol is shown in Fig. 1 using UML syntax: A protocol is formulated as a package template whose parameters are given in the upper right corner of the package. The package contains roles (with stereotype $\langle\langle role \rangle\rangle$) and classes. Roles constitute the participants in a protocol. They contain three compartments: attributes, operations and messages. Operations will be given by graph transformation rules. They manipulate attributes and exchange messages among roles. The rules of the contract net protocol are given in Figures 2 to 5. The global pre- and postconditions of the contract net protocol are specified by the goal rule in Fig. 6.

In order to avoid backtracking, the adoption of a protocol for a specific purpose in a given context is a decision taken at design time. In the model, it corresponds to the instantiation of a protocol template, whose parameters are filled in with model elements from a given context. Then, the model elements of the protocol are inserted in the agent class diagram and the role classes are attached to agent classes by a *role-of* relationship.

The graph transformation rules of the protocol are expanded in the following way: For each role in a rule an agent is inserted in the rule. The role and the agent are connected by the *role-of* relationship. The correlation of roles and agents must conform to the expanded agent class diagram. Then, model elements which

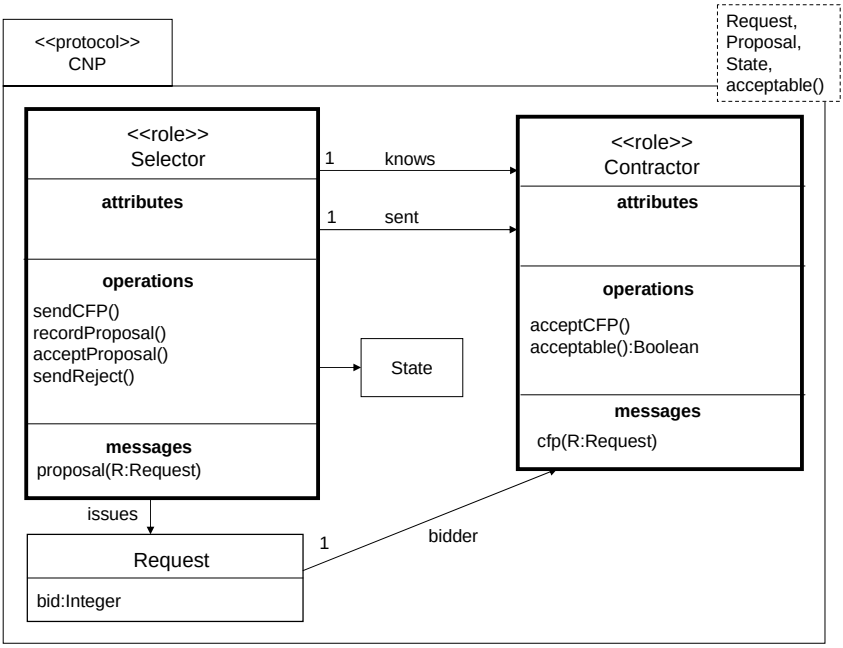


Fig. 1. The template of the contract net protocol

have been assigned to protocol parameters are inserted in the rule. Now, these expanded graph transformation rules reflect the change of the system behavior through the expanded protocol.

Thus, the operational behavior of an agent-based system is determined by the combination of (syntactic) template instantiation and (semantic) interpretation of the flattened model as a graph transformation system. This allows us to use the standard tools and theory of graph transformation.

Next, we exemplify the instantiation of the contract net protocol. In our example, an auctioneer and several customers participate in the English auction protocol. In order to use the contract net protocol, the selector role is attached to (assumed by) the auctioneer agent class and the contractor role is attached to the customer agent class. The parameters of the template are instantiated with the request and the proposal that are associated to the agent classes auctioneer and customer, respectively. In this way we obtain the diagram in Fig. 7. The agent classes contain two compartments: attributes and operations. Agents are not able to communicate without protocols. Therefore there is no compartment for messages. The graph transformation rules are also bound to the appropriate agents, see Fig. 8.

We have illustrated the syntax of our agent modelling language. A more detailed and formal description of the syntax is given elsewhere [5]. Next, we will focus on semantic aspects in order to describe the notions of goal and strategy more precisely.

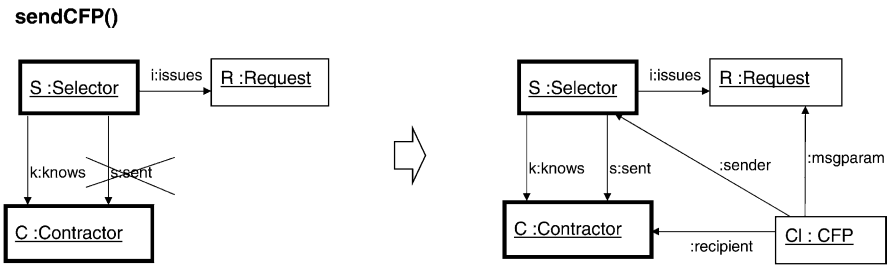


Fig. 2. The rule sendCFP()

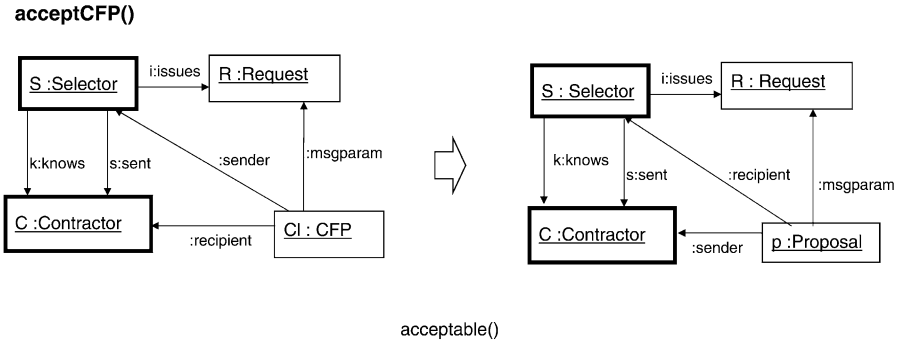


Fig. 3. The rule acceptCFP()

2.2 Formal Relationship between Goals and Strategies

In order to be able to verify whether strategies realize their goals, a formal definition of the relevant concepts is necessary. For this purpose we introduce some basic notions of graphs and graph transformations which serve as the basis for our formalization.

Typed graphs. In our models, graphs occur at two levels: the type level, given by a class diagram, and the instance level, given by all valid object diagrams. Such a type-instance relation is described more generally by the concept of *typed graphs* [4], where a fixed *type graph* TG serves as abstract representation of the class diagram. Its object diagrams are graphs G equipped with a structure-preserving mapping to the type graph, which is formally expressed as a *graph homomorphism* $tp_G : G \to TG$.

Rules and transformations. A *graph transformation rule* $p : L \Rightarrow R$ consists of a pair of TG-typed instance graphs L, R such that the intersection $L \cap R$ is well-defined. (This means that, e.g., edges which appear in both L and R are connected to the same vertices in both graphs, or that vertices with the same name have to have the same type, etc.) The left-hand side L represents the pre-conditions of the rule while the right-hand side R describes the post-conditions.

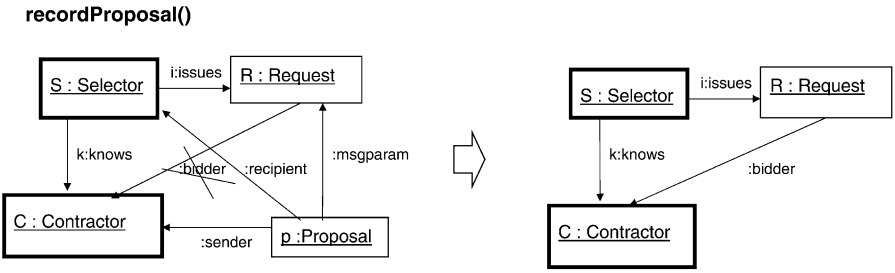


Fig. 4. The rule recordProposal()

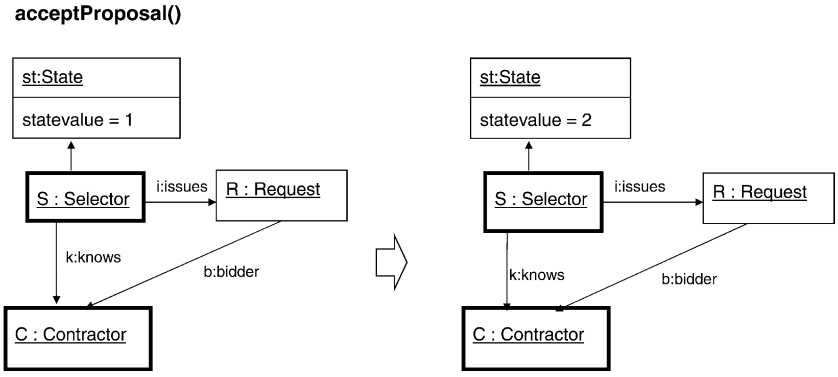


Fig. 5. The rule acceptProposal()

The application of a graph transformation rule is performed in three steps. First, find an occurrence o_L of the left-hand side L in the current object graph G. Second, remove all the vertices and edges from G which are matched by $L \setminus R$. Make sure that the remaining structure $D := G \setminus o_L(L \setminus R)$ is still a legal graph, i.e., that no edges are left dangling because of the deletion of their source or target vertices. In this case, the *dangling condition* [9] is violated and the application of the rule is prohibited. Third, glue D with a copy of $R \setminus L$ to obtain the derived graph H. We assume that all newly created objects, links, and attributes get fresh identities, so that $G \cap H$ is well-defined and equal to the intermediate graph D.

Graph transformation system. A *typed graph transformation system* $\mathcal{G} = \langle TG, C, R \rangle$ consists of a type graph TG, a set of structural constraints C over TG (like cardinality constraints, OCL expressions, etc.), and a set R of rules $p : L \Rightarrow R$ over TG. A transformation sequence $s = (G_0 \overset{p_1(o_1)}{\Longrightarrow} \cdots \overset{p_n(o_n)}{\Longrightarrow} G_n)$ in $\mathcal{G}$, briefly $G_0 \overset{s}{\Longrightarrow}{}^*_{\mathcal{G}} G_n$, is a sequence of consecutive transformations using the rules of $\mathcal{G}$ such that all graphs $G_0, \ldots, G_n$ satisfy the constraints C. The set of all transformation sequences in $\mathcal{G}$ is denoted by $\mathcal{G}^*$. As above, we assume that newly created elements are given fresh names, i.e., ones that have not been used before

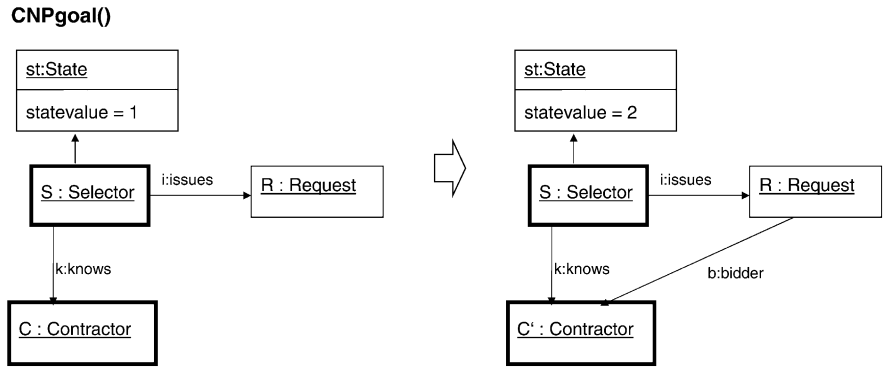


Fig. 6. The goal rule of the contract net protocol

in the transformation sequence. In this case, for any $i < j \leq n$ the intersection $G_i \cap G_j$ is well-defined and represents that part of the structure which has been preserved in the transformation from G_i to G_j.

We say that a transformation sequence $G_0 \overset{s}{\Longrightarrow}{}^*_{\mathcal{G}} G_n$ satisfies a rule $p : L \Rightarrow R$ if all the effects required by p are realized in the sequence. This is the case if there exists a graph homomorphism $o : L \cup R \to G_0 \cup G_n$, called *occurrence*, such that

- $o(L) \subseteq G_0$ and $o(R) \subseteq G_n$, i.e., the left-hand side of the rule is embedded into the pre-state and the right-hand side into the post-state, and
- $o(L \setminus R) \subseteq G_0 \setminus G_n$ and $o(R \setminus L) \subseteq G_n \setminus G_0$, i.e., at least that part of G is deleted which is matched by elements of L not belonging to R and, symmetrically, at the least that part of H is newly added which is matched by elements new in R.

With this formal background we can now define the relation of goals and strategies: A goal rule p should be implemented by a graph transformation system $\mathcal{G}$ given by a set of rules R, a type graph TG, and set of constraints C. Thus, all transformation sequences in $\mathcal{G}$ starting in a state that fulfills the pre-condition of p must lead to a state that fulfills the post-condition. This is captured in the following definition of a winning strategy.

Definition 1 (Winning strategy)
A strategy $\mathcal{G} = \langle TG, C, R \rangle$ is a winning strategy *for a goal rule $p : L \Rightarrow R$ if for all graphs G_0 that satisfy the constraints C, if the left-hand side of p occurs in G_0 via a graph homomorphism $o_L : L \to G_0$, then for all transformation sequences $G_0 \Rightarrow G_1 \Rightarrow \cdots \Rightarrow G_n$ in $\mathcal{G}$ there exists $i \in \{0, \ldots, n\}$ such that $G_0 \Rightarrow^*_{\mathcal{G}} G_i$ satisfies p via an occurrence $o : L \cup R \to G_0 \cup G_i$ extending o_L.*

In our running example, the graph transformation system $\mathcal{G}$ modelling the strategy is given by the class diagram in Fig. 1 representing the type graph TG, the protocol rules of Figures 2 through 5 representing the set R of rules, and

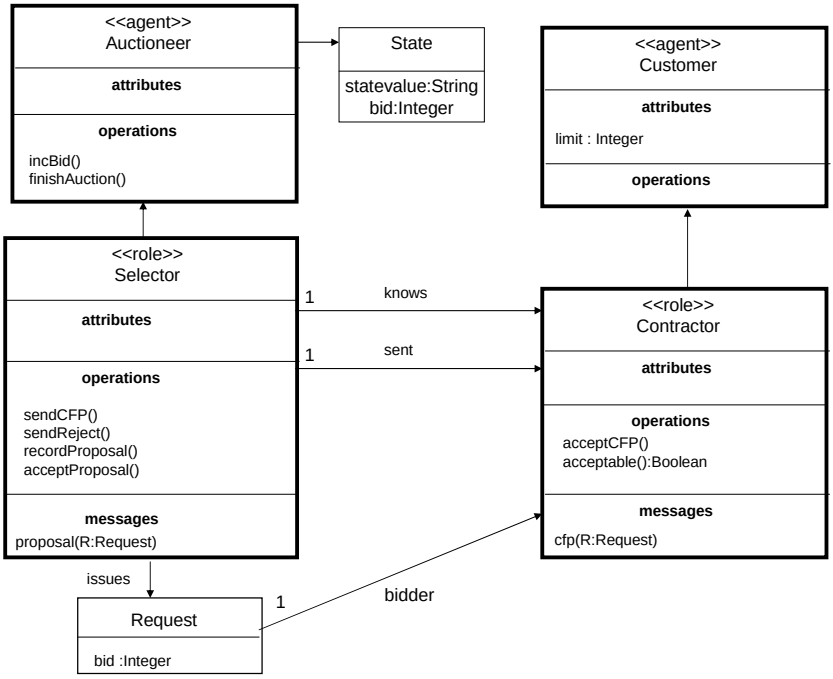


Fig. 7. Structural agent diagram

the and constraints C represented by the cardinality constraints in the class diagram. The goal rule p is given in Fig. 6.

Based on the formal interpretation of our language established in this section, the next section shows how the relationship between goals and strategies can be analyzed.

3 Model Checking Agent-Based Systems

Graph transformation rules can be seen as specifying a transition system whose states are graphs and whose transitions are graph transformation steps. For specifying and analyzing properties of such systems, temporal logic and model checking can be used. In this section, we explain how to derive a transition system from a strategy expressed by a set of graph transformation rules. Then, we show how temporal logic properties can be derived from goal rules, and how such properties can be analyzed using the Spin model checker.

3.1 From Strategies to Transition Systems

Recently, Varro has proposed an algorithm which maps graph transformation systems to state transition systems [14]. We adapt this algorithm for the trans-

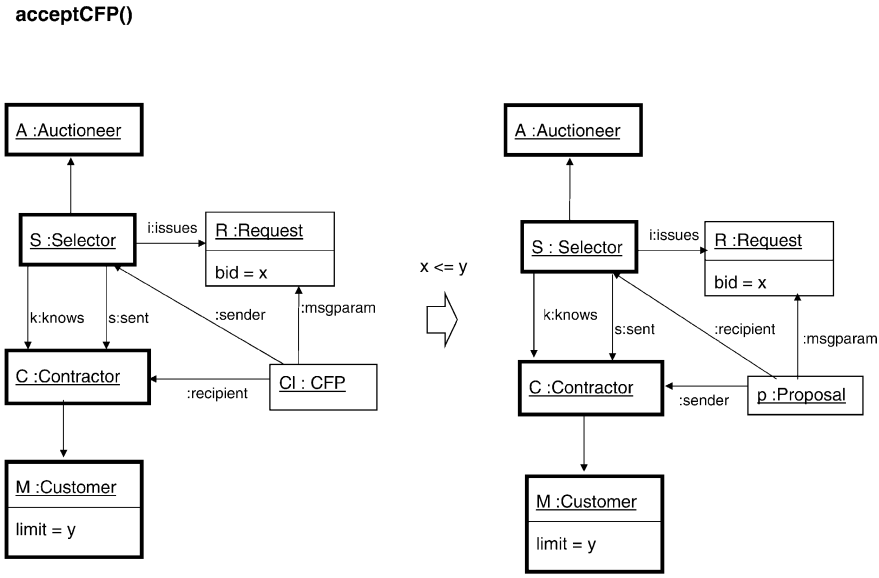


Fig. 8. Bound protocol rule acceptCFP()

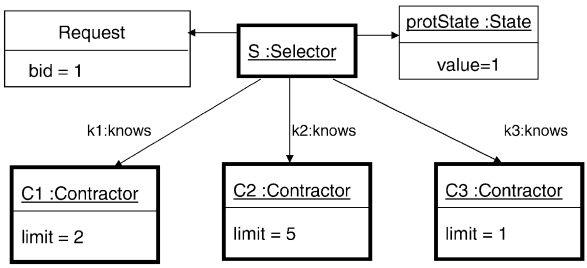


Fig. 9. Sample initial graph G_0

lation of a graph transformation system to the input language PROMELA of the SPIN model checker.

A strategy is given by a graph transformation system $\mathcal{G} = \langle TG, C, R \rangle$. Additionally, an initial graph G_0 is given representing the situation in which the strategy is applied. The state graph G_0 respects the constraints C of $\mathcal{G}$. The state graph G_0, the type graph TG and the rule set R are used to specify a transition system. With $\mathcal{G}$ as described in Section 2.2, a sample initial graph G_0 is shown in Fig. 9.

Graph transformation steps represent changes of concrete objects. This is in contrast with the abstract specification of these changes by means of rules. As a consequence, a single rule may lead to many different transformation steps which can be thought of as instantiations of this rule. In the operational semantics, this instantiation is performed by matching the left-hand side of a rule with the

current state graph, an operation that is not directly implementable in PROMELA or similar languages. Rather, the instantiation has to be done statically, as part of the translation, producing from every rule a set of transitions representing all possible instantiations of this rule. To make this possible, we have to sacrifice the dynamic nature of graph transformation for analysis purposes, limiting to a system with a fixed upper bound to the number of objects.

In the input language PROMELA of the model checker SPIN, states are represented by arrays of enumeration data types. To represent instance graphs, Boolean arrays are defined for all types (classes, agents, roles, associations, and attributes) in the type graph TG. The indices in these arrays represent the identities of the corresponding items.

In order to reduce the size of the state record, this representation is limited to such types whose instances are actually modified by a rule, the so-called dynamic part of the state. The static part, which is only read but never modified by any rule, is only represented explicitly as part of the initial graph G_0.

Transitions are specified as reaction rules in PROMELA. Reaction rules are parallel conditional assignments to state variables. For each graph transformation rule the pattern of the left-hand side is evaluated over the static part of the state. This results in a fixed number of transition specifications for each match of the pattern. The pre-condition of each corresponding PROMELA assignment contains a conjunction of atomic propositions on the dynamic state variables representing the matching process on the dynamic part. In the assignment part of the reaction rule, state variables are changed, e.g., representing the deletion of an object by setting its field from `true` to `false` or updating an attribute value, as specified by the graph transformation rule.

The sample graph transformation system $\mathcal{G}$ is transformed according to the following algorithm, adapted from [14].

```
ALGORITHM AgentGTS2STS
INPUT : graph transformation system G=<TG,C,R>, state graph G0
OUTPUT: PROMELA specification of a state transition system
METHOD:
BEGIN
     // Declarations
 1:  A 1D boolean state variable array
       for each agent, role and object class in TG
 2:  A 2D boolean state variable array for each association
 3:  A 1D enumeration type state variable array for each attribute
     // Initialization
 4:  Init arrays of rows 1 to 3 according to state given by G0
     // Encoding transformation steps as transitions
 5:  FOR EACH graph transformation rule p in R
 6:    generate guards for L, i.e., atomic propositions that test
         the validity of the elements of the left hand side of p
         with respect to the state variables
```

```
 7:    generate actions for R, i.e., assignments to the state
         variables that are to be changed according to
         the right hand side of p
 8:    generate a transactional transition specification, i.e.,
         an implication with a conjunction of the generated guards
         on the left hand side and a sequence of actions
         on the right hand side
 9:  END
END
```

Next, we show how the algorithm is applied to our running example.

State representation. Starting with the declaration of variables (rows 1 to 3 of the algorithm). G_0 consists of a number noOfContractors of contractors. The number of proposals is fixed to noOfProposals. In our sample state there are at most three contractors and three proposals. Boolean arrays cfp and prop are introduced for the dynamic objects of type Proposal and CFP which are generated or deleted by the application of graph transformation rules:

```
bool cfp[noOfContractors];
bool prop[noOfProposals];
```

All other instances of roles or objects are present in the start state already and are not generated or deleted during the execution of the protocol. The start state is also determined by the values of variables of static classes: a variable limit is introduced for each contractor role and the variable statevalue corresponds to the attribute of the object protState, see Fig. 9.

```
int statevalue;
int limit[noOfContractors];
```

For the dynamically changing links, further Boolean arrays are introduced:

```
bool sent[noOfProposals];
bool cfpSender[noOfProposals];
bool cfpRecipient[noOfProposals * noOfContractors];
bool propSender[noOfProposals * noOfContractors];
bool propRecipient[noOfProposals];
bool bidder[noOfContractors];
```

All arrays together form the state representation of the system, which we investigate with respect to the contract net protocol. The arrays are initialized (row 4 of the algorithm) according to the start state graph G_0, see Fig. 9. For example, the limit values are set and the sent links are initialized to false:

```
statevalue=CNPstart;  /* CNPstart = 1, CNPend = 2 */
limit[1]=2;
limit[2]=5;
...
```

```
sent[1]=false;
sent[2]=false;
...
```

Transitions. For rule sendCFP (cf. Fig. 2) we obtain one reaction rule for the match given by the contractor role 1 of role type Contractor and the message 1 of type cfp:

```
/* sendCFP */
:: atomic { msgreceived[1]==false ->
        msgreceived[1]=true; cfp[1]=true; cfpSender[1]=true;
        cfpRecipient[1 + 1 * noOfContractors)]=true;         }
```

The model we are going to analyze has now been translated into the appropriate transition system specification. In the following, we will see, how the properties to be analyzed can be derived from the goal rule.

3.2 Expressing Goals in Spin

The next task is to translate to SPIN the notion of a winning strategy for a goal rule $p : L \Rightarrow R$. The left-hand side L is a condition for the initial state of the system. If this condition is satisfied, for a winning strategy we require that the right-hand side R will be eventually satisfied in every run of the protocol. However, since L and R are not disjoint, the occurrences of L and R should coincide on the shared items (cf. Section 2.2). In order to translate this condition, for each occurrence of L, we must take into account all occurrences R in the reachable states of our model.

In PROMELA, properties of models can be expressed as correctness claims. They are built up from simple propositions, given by Boolean conditions on the state variables, and temporal operators. Next, we are constructing a temporal formula from a goal rule and express the same as a correctness claim.

Like the translation for strategies, we are mainly concerned with the changing parts of the state, i.e., only the changing parts of a goal rule need to be matched dynamically while the invariant parts just have to be matched in the initial state. The latter is done at compile time, producing separate formulae for every match which are then combined by disjunction.

Thus, for every occurrence $o_L(L)$ in our initial graph G_0, a compatible later occurrence $o_j(R)$ is required. This is expressed by a temporal formula in which the "later occurrence" is required by the temporal operator $\diamond$ (eventually):

$$goal_i = (o(L) \Rightarrow \diamond(o_1(R) \vee \ldots \vee o_k(R)))$$

Note that all different occurrences $o_1, \ldots, o_k$ of R are enumerated which satisfy the compatibility with the choosen occurrence o_L of L.

The formulae for all different occurrences of L are connected in a disjunction:

$$goal = (goal_1 \vee \ldots \vee goal_n)$$

In our example the disjunction only has a single member because there is only one match of the rule p (see Fig. 6) in the start state G_0 (see Fig. 9). The right hand side of p matches to three different contractor role occurrences. Note, that the contractor roles belong to the invariant parts of the state.

In PROMELA we have to express the matches in terms of the variables which were introduced in the previous section along with the translation of strategies. Again, only the dynamic parts of the matches have to be considered:

```
#define OccL (statevalue==CNPstart)
#define OccR1 (statevalue==CNPend && bidder[con1]==true)
#define OccR2 (statevalue==CNPend && bidder[con2]==true)
#define OccR3 (statevalue==CNPend && bidder[con3]==true)
```

From this definitions a temporal logic formula is built which serves as input to the model checker SPIN:

```
#define CNPgoal (OccL --> <>(OccR1 || OccR2 || OccR3))
```

The checking of the sample communication strategy, the contract net protocol, with respect to the property CNPgoal has been successful. This means that our strategy realizes the goal in the situation given by G_0, but is not enough to prove that $\mathcal{G}$ is indeed a winning strategy with respect to p.

4 Related Modelling and Model Checking Approaches

We consider approaches related to different aspects of our proposal to the modelling and analysis of agents' goal-driven behavior.

AgentUML is an example for the extension of UML with language elements for modelling agent concepts [3], but the extension is only defined informally: It does not use the extension mechanism of UML nor does it describe the extension at the level of abstract syntax, i.e., the meta model. Also, AgentUML lacks a formal semantics which reflects agent concepts precisely.

With respect to model checking there is an approach by Wooldridge et al. in which the model checker SPIN is used for verifying the behavior of agent systems [16]. Different to our approach, the behavior description relies strongly on a kind of BDI-logic which is translated both to PROMELA-models and to temporal claims in SPIN.

The *Mocha* approach to model checking agent based systems is a proposal and tool for modular model checking [1]. Generally, a system consists of more or less loosely coupled parts which are integrated. A state transition graph of the whole system forms the model to be analyzed. Since model checking is based on exhaustive state-space exploration, and the size of the state space of a design grows exponentially with the size of the description, scalability is an important issue.

Mocha introduces a hierarchical modelling framework of interacting and reactive modules. Thus, it avoids the manipulation of unstructured state-transition

graphs. But this also happens in our case: each rule of a strategy can be considered as a goal rule which is to be refined by a strategy. In this way the model checking of goal-reachability is also decomposed hierarchically. Because local variables within a strategy do not influence the reachability of other goals, the hierarchical decomposition of goals and strategies also reduces the state-space to be explored.

For requirements specification in Mocha, the languages of linear- and branching-time temporal logics are replaced with *Alternating Temporal Logic* (ATL). This allows to express both cooperative and adversarial requirements, e.g., that a module can attain a goal regardless of how the environment of the module changes. These features of ATL compare to our analysis of goal-reachability. External parameters of a goal rule are, for example, the roles of the agents not to be analyzed. If we do not fix these parameters then the results of the model checking answer the question, whether an external element influences the reachability of a goal.

All related approaches to agent-oriented model checking use textual modelling languages. They essentially support the specification of state transition systems and the formulation of logical formulas for requirements specification. In contrast, we showed how high-level visual specifications based on UML and graph transformation rules, and requirements given by goal rules can be analyzed.

5 Conclusion

Objects and agents differ in the way they cope with requirements. Objects are supposed to comply with the requirements that are given more or less formally in a requirements specification document. Different to objects, agents may be *controlled* by requirements, represented as goals. Agents are striking for their goals by applying appropriate strategies.

In this paper we have dealt with the modelling of agents' behavior. The question is, how can goals and strategies be modelled appropriately and how can a strategy be verified to attain a goal. We have focused on communications strategies, i.e., protocols for the interaction of agents. For solving the problem, we introduced graph transformation as a semantic domain for the visual modelling of agents: Single graph transformation rules are considered as goals and a graph transformation system specifies a protocol by behavioral rules for all participating agents.

The applicability of graph transformation rules depends on the current structure of the system state. Therefore the question, whether a communication strategy attains a certain goal can be answered only for a given initial state and restricting the number of objects to a ensure a finite state space. With these assumptions, we have translated a model to a transition system specification in Promela, the language of the Spin model checker. We have shown how a goal rule can be translated to an appropriate temporal logic formula. Then, the reachability of the goal could be analyzed automatically in Spin.

In this paper, we have limited ourselves to a single level of refinement of a goal by a strategy. In turn, each rule of a strategy can be considered as a subgoal which is to be refined by another strategy. The recursive substitution of a goal rule by a set of rules requires that the subsequent application of these rules has no additional effects beyond those that are specified by the goal rule. The conditions for the validity of this frame condition have to be investigated in the future.

References

1. R. Alur, L. de Alfaro, R. Grosu, T. Henzinger, M. Kang, R. Majumdar, F. Mang, C. Kirsch, and B. Wang. Mocha: A model checking tool that exploits design structure. In *Proc. 23rd Intl. Conference on Software Engineering (ICSE'2001)*, pages 835–836, Los Alamitos, CA, 2001. IEEE Computer Society Press.
2. P. Baldan, A. Corradini, H. Ehrig, M. Löwe, U. Montanari, and F. Rossi. Concurrent semantics of algebraic graph transformation. In H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 3: Concurrency and Distribution*, pages 107 – 188. World Scientific, 1999.
3. B. Bauer, J.P. Müller, and J. Odell. Agent UML: A Formalism for Specifying Multiagent Software Systems. In P. Ciancarini and M.J. Wooldridge, editors, *Proc. 1st Int. Workshop on Agent-Oriented Software Engineering (AOSE 2000), Limerick, Ireland, June 2000*, volume 1957 of *LNCS*, pages 91–104. Springer-Verlag, Berlin, 2001.
4. A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26(3,4):241–266, 1996.
5. R. Depke, J.H. Hausmann, and R. Heckel. Design of an agent-based modeling language based on graph transformation. In M. Nagl and J. Pfaltz, editors, *Int. Workshop on Applications of Graph Transformations with Industrial Relevance (AGTIVE 2003)*. LNCS. Springer-Verlag, September 2003. To appear.
6. R. Depke, R. Heckel, and J. M. Küster. Roles in agent-oriented modeling. *Int. Journal of Software Engineering and Knowledge Engineering*, 11(3):281–302, 2001.
7. R. Depke, R. Heckel, and J. M. Küster. Formal agent-oriented modeling with graph transformation. *Science of Computer Programming*, 44:229–252, 2002.
8. G. Weiss (ed.). *Multiagent Systems*. MIT Press, Cambridge, MA, 1999.
9. H. Ehrig, M. Pfender, and H.J. Schneider. Graph grammars: an algebraic approach. In *14th Annual IEEE Symposium on Switching and Automata Theory*, pages 167–180. IEEE, 1973.
10. Foundation for Intelligent Physical Agents (FIPA). Agent communication language. In *FIPA 97 Specification, Version 2.0*, `http://www.fipa.org`. FIPA, 1997.
11. N. R. Jennings and M. J. Wooldridge, editors. *Agent Technology: Foundations, Applications, and Markets*. Springer-Verlag, 1998.
12. Object Management Group. UML specification version 1.5, March 2003. `www.omg.org`.
13. R. G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. In *IEEE Transaction on Computers*, number 12 in C-29, pages 1104–1113, 1980.
14. D. Varro. Towards automated formal verification of visual modeling languages by model checking. *Journal of Software and System Modeling*. To appear.

15. M. J. Wooldridge, N. R. Jennings, and D. Kinny. The Gaia methodology for agent-oriented analysis and design. *Journal of Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.
16. Michael Wooldridge, Michael Fisher, Marc-Philippe Huget, and Simon Parsons. Model checking multi-agent systems with MABLE. In Maria Gini, Toru Ishida, Cristiano Castelfranchi, and W. Lewis Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'02)*, pages 952–959. ACM Press, July 2002.

Giving Life to Agent Interactions*

Juliana Küster Filipe

Laboratory for Foundations of Computer Science
School of Informatics
The University of Edinburgh
King's Buildings
Edinburgh EH9 3JZ
United Kingdom
jkfilipe@inf.ed.ac.uk

Abstract. Agent UML (AUML) is an extension of the standard object-oriented modelling language UML adapted for agent-based systems. In AUML sequence diagrams are extended to describe specific dynamic aspects of agents including dynamic role assignment, nondeterministic choice in agent decisions and concurrent communication. These diagrams are used to describe agent interaction protocols. We describe some of the problems of extended sequence diagrams in AUML. Not all of these problems are specific to agent sequence diagrams and have been previously identified in UML's sequence diagrams and Message Sequence Charts (MSCs). Live Sequence Charts (LSCs) are a very expressive extension of MSCs that addresses many of these issues in the context of object-oriented systems. We propose an extension of LSCs for agents, and show how it is useful for modelling agent interactions, agent commitments, agent society norms and forbidden behaviour. Further, we show how to derive formulae in a simplified variant of the distributed temporal logic MDTL from an extended LSC.

1 Introduction

As agent-based approaches are gaining acceptance in the software development community it becomes fundamental to make well-defined languages available to software engineers and consequently promote principled methodologies for system development. One promising way of doing so is by linking agent-based methods to more commonly understood object-oriented technologies and extend these as needed to cover agent-specific aspects.

In recent years, UML [29] has become the standard modelling language for object-oriented system design and because many tools promote (essentially only the documentation of) design in UML it has become widely used in industry. Work has been done on extending UML diagrams and notation to support typical agent aspects including the development of Agent UML (AUML) [25, 2, 3, 5, 13]. AUML is also being developed further in collaboration between the Object Management Group (OMG) and FIPA (Foundation for Intelligent Physical

* Work reported here was supported by the EPSRC grant GR/R16891.

M.D. Ryan et al. (Eds.): Objects, Agents, and Features, LNCS 2975, pp. 98–116, 2004.

Agents) [15] to address the changes being made to UML2.0 [28]. Another non-UML based approach for the analysis and design of agent-oriented systems which influenced AUML considerably is the Gaia methodology [30].

There are many different definitions of agents, and in particular many attempts to characterise what distinguishes agents and objects. An interesting analysis of the differences can be found in [24]. The confusion between objects and agents often arises from the fact that simple agents can look very much like objects. The idea behind modelling with agents is nonetheless rather different from modelling with objects. Agents are an abstraction for decentralised problem solving, whilst objects are a structuring mechanism useful for programming. Agents are good for solving problems, objects are useful for programming systems. To some extent agents are at a higher level of abstraction than objects.

When modelling agent-based systems, it is common to distinguish between a *micro* and a *macro* view. The micro view concerns local aspects of individual agents including strategies, knowledge, beliefs and behaviour. The macro view considers how agents interact and communicate with each other and with the environment. In OOA/OOD interactions are commonly modelled using scenario-based languages like Message Sequence Charts (MSCs)[31] or UML's variant of MSCs called sequence diagrams.

In AUML sequence diagrams are extended to describe specific dynamic aspects of agents including dynamic role assignment, nondeterministic choice in agent decisions and concurrent communication. These diagrams are used to model agent interaction protocols describing allowed communication patterns, that is patterns of messages sent from one agent role to another. In this paper, we describe the problems of extended sequence diagrams in AUML. As an example, we cannot describe *disallowed* communication patterns in AUML.

Some of the problems of agent sequence diagrams are not specific to the agent extension itself and are inherent to sequence diagrams (and message sequence charts) in general, but others arise from the agent extensions. Solutions to these problems are needed in order to be able to define a formal semantics. A fully worked out semantics is important for automatic synthesis and consistency checking. By synthesis we mean the ability to automatically construct a behavioural equivalent state-based specification (for example statecharts) from the scenarios (given by the sequence diagrams). This is a necessary first step towards moving automatically to implementation. Without such prospects a modelling language has limited use.

Live Sequence Charts (LSCs) [4] are a very expressive extension of MSCs that addresses many of the problems of MSCs and UML's sequence diagrams in the context of object-oriented systems. In this paper, we propose a simple extension of LSCs for agents. Our extension is very useful when modelling agent interactions, agent commitments and contracts, as well as society norms and forbidden behaviour. One of the major advantages of LSCs is that while they consist of a visual notation which appeals to engineers they have a formal semantics and the expressiveness of temporal logic. We see how from an LSC we can derive formulae in a simplified variant of the branching time logic MDTL.

This paper is structured as follows. In the next section we describe how agent interactions can be modelled in AUML and point out some of the weaknesses. In Section 3 we describe Live Sequence Charts and an extension thereof for agents. Several examples illustrate how to use the extension for modelling agent interactions, agent commitments, agent society norms and forbidden behaviour. In Section 4 we briefly describe some semantic aspects of the extension and show how to move from the visual representation to logical formulae. The paper finishes with some conclusions and ideas for future work.

2 Modelling Interactions

One of the major concerns of AUML has been to extend UML in order to model agent interaction protocols. Agent interaction protocols describe allowed communication patterns, that is patterns of messages sent from one agent role to another. The subset of UML of interest here are sequence diagrams.

Sequence diagrams are one of the two forms of interaction diagrams available in UML. Interaction diagrams (both sequence diagrams and collaboration diagrams[1]) are used to capture how instances interact in order to perform a specific task which the individual instances could not perform on their own. Each diagram shows one interaction (per scenario) as a collection of communications between instances partially ordered over time. A message is a communication between instances. It can cause an operation to be invoked, a signal (asynchronous) to be raised, an instance to be created or destroyed.

Sequence diagrams are an extension of Message Sequence Charts (MSC) [31]. Like MSC they represent time explicitly making the order in which messages are sent visually more intuitive. There are two dimensions in a sequence diagram: a vertical dimension denoting time; and a horizontal dimension representing the different instances involved in the interaction. Each instance participating in the interaction has a vertical lifeline (a dashed line under the instance symbol representing the existence of the instance at a particular time). Messages are shown as horizontal/oblique labelled arrows between the lifelines of the instances. Arrow labels contain the invoked operation or signal as well as optional conditions and/or iteration expressions. Instances can be created or destroyed which is shown intuitively by the fact that a lifeline starts or stops. Constraints and notes can be attached to the model (e.g., for indicating timing constraints, durations, and so on).

At a very high level sequence diagrams can be used to realise use cases, that is to describe the interactions intended by a use case between the actors (external entities to the system, e.g., users or other systems) and the system itself. Sequence diagrams can be further refined throughout development as interactions get elaborated.

When used in an object-oriented context the instances mentioned above normally correspond to instances of classes, that is the objects that will exist in the system at runtime. In an agent-oriented context sequence diagrams can be used

[1] Collaboration diagrams have been renamed *communication* diagrams in UML 2.0.

to describe interactions between agents or at a specification level between agent roles.

In AUML sequence diagrams are extended in order to capture two fundamental and distinctive aspects of agents:

1. *roles*: agents can play one or more roles at a given moment in time and throughout their life time. In particular, agents can change roles dynamically.
2. *concurrency*: agents have internal concurrency, that is, agents can do several things at the same time including participating in different conversations with several agents.

Consequently, sequence diagrams have been extended to allow for multiple roles, and multiple concurrent threads [25, 2]. In the first case, stereotyped messages are used to indicate agents changing roles dynamically (including switching between roles, taking up a new role, and so on) but we omit further details [26]. We are particularly interested in the notation used for inter-agent communication and concurrent communication. Consider the notation given in **Fig. 1** which is not possible in UML1.x.

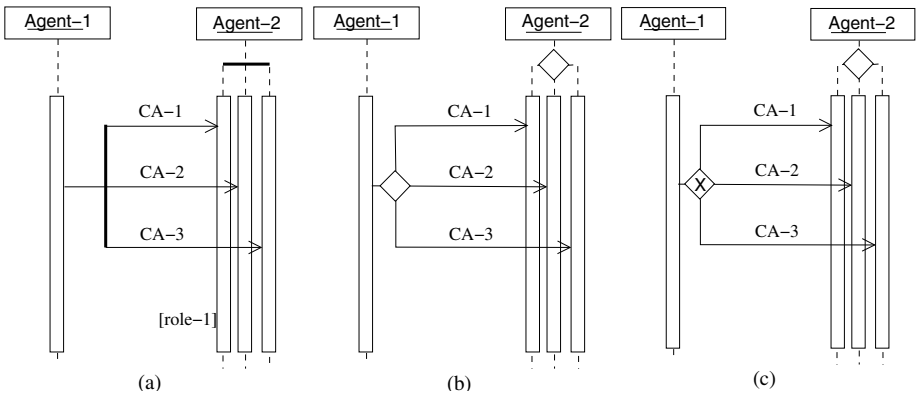


Fig. 1. Different notations for inter-agent communication.

All communication is asynchronous in the different cases as is indicated by the stick arrowhead. Synchronous communication is normally not used between agents in AUML but indicated in UML by a filled solid arrowhead. Case (a) shows agent-1 sending multiple messages concurrently to agent-2. The messages sent denote *communication acts* in agent terminology and are given by CA-1, CA-2 and CA-3 (or any other finite number). Agent-2 processes these messages concurrently as well. In fact, agent-2 may be playing different roles at the same time and each role be processing one message independently. Roles are indicated in square brackets next to the corresponding vertical lifeline (or thread). In case (b) the diamonds indicate a decision box meaning that agent-1 may *decide* to send zero or more (at most three in this case) messages to agent-2. If more than

one message is sent then these are sent concurrently. Consequently, if agent-2 receives more than one message it processes them concurrently. Again agent-2 may be playing different roles while doing so. The empty diamond denotes an inclusive or. Finally, in case (c) the x-labelled diamonds denote an exclusive or. Agent-1 decides to send only one of the several possible messages to agent-2. The exclusive or denotes nondeterminism. Notice that in UML1.x we can express conditional branching (if-then-else) which is, however, not the same thing as an exclusive or.

There is a common alternative notation to each of the three cases above which makes the diagram look less cluttered and is useful when several communications are combined in one diagram. **Fig. 2** shows case (b) in the alternative notation. Nonetheless this notation is ambiguous and misleading. Since the activation

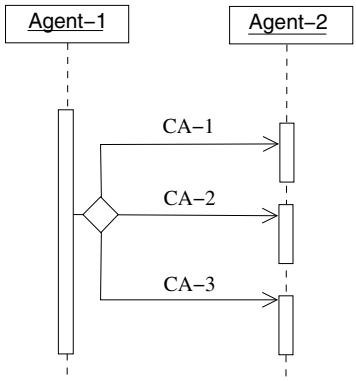


Fig. 2. Alternative notation to the inclusive or case.

bars are placed along a lifeline we loose the visual intuition that the events of agent-2 corresponding to the receive of CA-1 and CA-2 are either concurrent and correspond to different agent roles, or in conflict and not part of the same system run. We may erroneously be led to think that the events are happening sequentially.

In UML1.x instances have their name underlined. For example `o/Role:C` denotes an object `o` of class `C` playing the role `Role`. This can be easily adapted for agents, for example, `a/Role` can be used to denote an agent `a` playing role `Role`. When we only indicate the role names instead of instances we do not need to underline them. Since we can have several agents playing a given role, it is common in AUML to add multiplicities to the messages to indicate when a message is explicitly being sent to more than one agent.

Consider the sequence diagram given in **Fig. 3** describing the so-called English auction interaction protocol. The diagram is essentially that used in [14] (with some minor corrections and additions in accordance to the same protocol

as given in [2]) where it is given generically as a template describing a communication pattern. We omit the template notation here for simplicity.

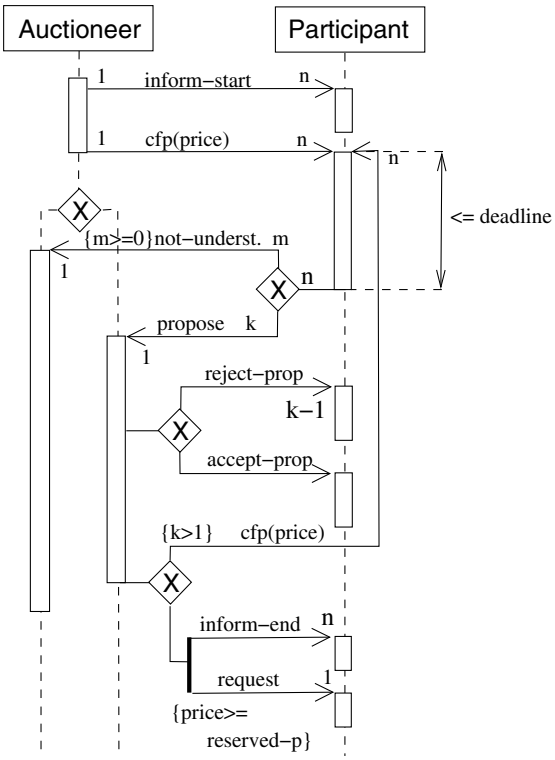


Fig. 3. FIPA English Auction Interaction Protocol.

In an English auction, the auctioneer tries to sell an item by initially proposing a price below the supposed market value and then gradually raising the price. Each time the price is announced, the auctioneer waits to see if any buyers will signal their willingness to pay the proposed price. As soon as one buyer indicates that it is willing to accept the price, the auctioneer issues a new call for bids with an incremented price. If no participant is prepared to pay more, the item is sold for the last offered bid as long as the offered price is more than the reservation price. The auctioneer informs the participants that the auction is over.

What is described in the sequence diagram in **Fig. 3** is the following. The auctioneer notifies the participants that the auction begins (`inform-start`) and announces an initial price (`cfp` - call for price). Both messages have multiplicities 1 to n, where n is a finite number of agents playing the role of participants in the auction. The problem of using an n here is twofold. Firstly, it does not conform with the usual practice in UML1.x for indicating multiplicities, and secondly it

imposes that the number of agents in the auction room is the same at all times. If an agent is "late" and enters the auction room after the inform-start has been sent, then it will apparently not receive any of the messages sent afterwards (e.g. `cfp`). A $\star$ would solve this problem, but to some extent not ease what is intended in the next communication where an n is next to the decision box.

Before the deadline, each participant agent decides to reply either that it did not understand or that it accepts the bid (indicated by the message propose). The multiplicities at this point seem to indicate that all participants reply to the auctioneer and decide either to say that they do not understand or that they accept the price. Supposedly, m indicates the number of agents that decide not to understand, whilst k stands for the number of agents that reply that they accept the price. Consequently, we are to assume that $k + m = n$.

The auctioneer then decides to either accept or reject a bid. Again, the multiplicities indicate that the auctioneer accepts one bid and rejects all the others. However, it is not possible to tell which participant corresponds to the accepted bid. The decision box will allow the auctioneer to choose one as desired.

Later, the auctioneer decides to either continue the auction by sending another `cfp` (if on the last bid more than two participants showed an interest), or to end the auction. If the auctioneer decides to end the auction it sends a message to all participants that the auction has ended and (concurrently) sends a message to the last bidder if the price offered is indeed higher than the reserved price. It is not possible to tell how to refer to the agent that made the last accepted offer. Basically this sequence diagram would allow for a situation in which a participating agent would receive a message indicating that the item was sold to it, even though that agent did not bid for the item. Furthermore, even if $k \geq 1$ the auctioneer may still decide to end the auction which is not intended.

We have already identified a few points in the diagram that can be misleading. Other important issues are:

1. Does the sequence diagram describe *one* possible scenario in an English auction place, or does it constrain *all* possible system runs?
2. Attached to the first decision box there is a multiplicity n. Does the given multiplicity enforce the number of messages sent, i.e., does it mean that n agents *must* or *may* reply in one of the two possible ways? Since agents are autonomous and take their own decisions it is reasonable to have a *may*, but on the other side some society norms (in this case a norm imposed on agents attending an auction) could impose a *must* (regardless of whether the agents indeed conform to the norms or not). It therefore seems important to be able to express what is intended in the diagram.
3. Can messages be lost? For example, a participant may send a proposal which the auctioneer never receives.
4. How can we express that the auctioneer will accept the first bid received? How can we send a message to a particular agent in a set?
5. How can we guarantee that eventually the auctioneer will decide to end the auction? Even if $k = 1$ the auctioneer may never sent a message inform-end (we cannot guarantee that any of the roles in the diagram are progressing).

6. Is it possible to say that a certain behaviour is forbidden? This is particularly useful if we want to describe society norms. Agents may then still decide not to conform to these norms but the agent's are then explicitly violating one or more norms.

It is not possible to express any of these issues in the extended sequence diagrams for agents. Some of these issues are not particular to the agent extension itself and are inherent to sequence diagrams (and message sequence charts) in general, but others arise from the agent extensions.

An attempt to improve the expressiveness of extended sequence diagrams in AUML by defining new *message stereotypes* is given in [20]. The English auction interaction protocol is also described in the paper with the new notation. In particular, one of the new message stereotypes can be used to denote a message that *must* be received. However, most of the issues mentioned above are still unsolved.

In the next section we describe Live Sequence Charts (LSCs) [4, 18] and describe an extension for modelling agent interactions. Like sequence diagrams, LSCs are a visual scenario-based approach. But LSCs are more suitable for our purposes because they address most of the issues above. A simple extension makes LSCs adequate to describe agent interactions as needed.

AUML notation is currently changing in order to accommodate the changes being made to sequence diagrams in UML2.0 [28]. Curiously, UML2.0's sequence diagrams have received substantial influence by LSCs themselves. Not all issues seem to be dealt with though. In particular, sequence diagrams in UML2.0 continue to only consider an interleaving semantics (even though they allow for parallel processing), whilst for agents we believe that true concurrency is more natural (at a high level at least). Information on the changes to AUML can be found in a recent working draft [15].

3 Live Sequence Charts for Agents

In the previous section we have described some of the problems of the extended sequence diagrams in AUML. Sequence diagrams are used to model how agents interact but cannot be used to explicitly state mandatory, possible or forbidden behaviour. This is important when modelling norms or rules in an agent society or multi-agent system.

An extension of MSCs addressing exactly these problems is called Live Sequence Charts (LSC)[4]. The motivation behind the name comes from the fact that these charts make it possible to express *liveness* constraints, that is, that eventually something happens enforcing progress of the instances along their lifeline.

There are also a few interesting extensions of LSCs adding time, symbolic instances to allow for generic scenarios, assignments, conditions and simultaneous regions [17, 22, 16, 21]. The original paper on LSCs already allowed so-called *coregions* which correspond to regions in the charts where the events are unordered. However, it is only in [21] that coregions and explicit simultaneous

regions are given a true-concurrent interpretation. A true-concurrent semantics is what we want for modelling agent behaviour as well.

In this paper, we do not provide a complete characterisation of LSCs. For a recent, detailed presentation of LSCs and their (partial) implementation in a tool called the Play Engine see [18]. Here, we explore an extension of LSCs for agents.

An LSC can describe a scenario that *may* happen (existential) or that *must* happen (universal). The distinction between may and must comes at several levels: messages that may/must be received, time that may/must progress along a lifeline of an instance, and conditions that may/must hold. May elements are *cold* whilst must elements are *hot*. Notice that unlike sequence diagrams, conditions in LSCs are interpreted (they carry a semantic meaning and are not just annotations). In particular, conditions may correspond to behaviour. For example, conditions can be used to describe scenario(s) which if successfully executed force the system to enter the actual body of the chart. If used in such a way, the condition denotes a chart precondition and is called prechart. This is very useful as we will see later in an example.

The body of an LSC may contain further subcharts. A subchart can be of three kinds: a prechart (as mentioned above), a loop or an if-then-else. Loops and if-then-else subcharts are used in combination with conditions. Notice that, using a cold condition in the beginning of an unbounded loop corresponds to a *while* loop, whilst using a cold condition in the end of a loop leads to a *repeat-until*. When a cold condition does not hold the subchart is exited. If on the contrary a hot condition does not hold then the whole chart is aborted. A hot condition *must* be true, otherwise the requirements are violated. When used properly hot conditions can describe *forbidden* behaviour. For example, if we want to forbid a certain sequence of communications, we specify these communications in the prechart of an LSC and place a hot false condition in the body of the chart (see example in **Fig. 5**).

In terms of notation, all hot elements are given in solid lines/boxes/hexagons whereas cold elements are indicated in dashed lines/boxes/hexagons. Conditions are shown in hexagons (so are precharts since they denote preconditions) whilst assignments are given in a box with a turned top right corner (similar to notes in UML). If within a lifeline (normally dashed - notice that in LSCs there are no activation bars) the line becomes solid it means that the instance must proceed along the lifeline. By contrast, a dashed lifeline means that the instance may proceed according to the specified behaviour but it does not have to. *Dotted* lines are used for coregions.

The expressive power of LSCs is very close to what we need for modelling agent interactions. Our proposed extension to LSCs is therefore simple and along two lines:

1. We introduce *decision* subcharts including *inclusive or* subcharts and *exclusive or* subcharts to achieve what extended sequence diagrams in AUML express with the diamond notation (see **Fig. 1**).

2. We allow *explicit* message sending and *arbitrary* message sending. By *explicit* message sending we mean that an agent sends a message to a specific agent or set of agents it knows about. It is indicated using OCL2.0's [27] message notation in the message label. By *arbitrary* message sending we mean that there is no specific target and any agent playing the receiver role (at the moment in time that the message is received) will or may receive the message[2].

The first point is an extension to LSCs. Notice that it is possible in LSCs to describe nondeterministic choice using a condition on subcharts containing a *SELECT* basic expression. This expression comes associated with the probability the condition evaluates to true or false. However, it is not always possible to associate quantitative information to agent decisions and therefore use probabilities. Moreover, it would certainly not be adequate for *inclusive or* cases. The case (a) of **Fig. 1** reflecting concurrent communication can be expressed in LSCs using coregions.

The second point is needed essentially because we use LSCs at a specification level rather than at an instance level. Recall that LSCs are normally used for describing object interaction and each object involved in a scenario has its own lifeline. Messages are therefore only sent from one object to another. When describing communication between roles message sending becomes more complex.

In OCL2.0, `a.b^msg()` means that object `a` sends a message `msg()` to object `b` (here we can have agents as well). If the sending object/agent is clear (or there is at most one agent playing the sender role) then `a` can be omitted. We can also write `a.B^msg()` where `B` is a set of agents `a` knows about. Notice that this notation is only needed if there are many agents possibly playing the receiver role. Further, we use OCL in all assignments and conditions. Notice that using OCL in message labels simplifies the notation avoiding message multiplicities as in AUML and allows us to send messages to particular agents in a set. By default all messages can be received by any agent playing the role of the receiver.

We illustrate the notation of LSCs and our extension in **Fig. 4** using the example of the English auction interaction protocol introduced in **Fig. 3**. To simplify, time is not considered here. Notice, however, that we can express time in LSCs [17] and adding timing constraints here is straightforward. We will see time in an example later on.

As before, we have two agent roles: Auctioneer and Participant. The chart starts with a prechart where there is one cold condition on the lifeline of Auctioneer (conditions can sometimes affect several instances or roles in which case they cross all the affected lifelines) stating that there is at most one agent playing the role of auctioneer (this is OCL notation). After the auctioneer sends a message indicating the start of the auction (this message could possible be lost) and the

[2] Notice that explicit message sending corresponds to either *point-to-point* communication or *multicast*, whilst arbitrary message sending corresponds to *broadcast*. For the latter, we assume that all agents playing the receiver role at the time the message is sent may receive it, regardless of whether an agent decides to ignore received messages by "not listening".

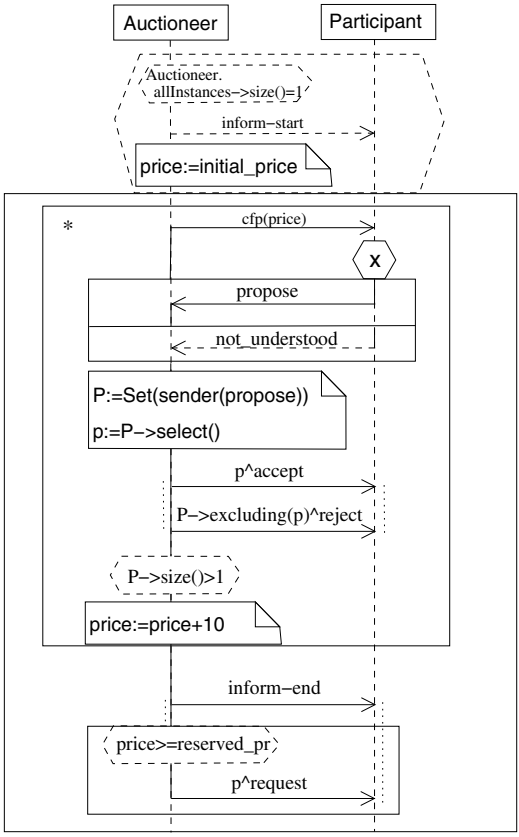


Fig. 4. An extended LSC for the English Auction Interaction Protocol.

auctioneer sets the price, the body of the chart is entered and consequently the loop subchart (loops are indicated by a $\star$ at the top left corner).

In the loop, the auctioneer sends a message to the participants with a `cfp`. All participants in the auction room receive the message and must decide (to indicate a may decide we would have a dashed hexagon) whether to accept the bid or send a message `not_understood`. The auctioneer knows the agents that accepted the bid (set P), and selects one of them. Notice that there is deliberately no argument in the `select()` operation because we are assuming that only the auctioneer knows what the selection criteria is. Thereafter, the auctioneer must send two messages concurrently: one to the selected participant accepting the offer, and one (or more) to all the remaining participants that proposed as well rejecting their offer. The concurrency is indicated by the coregion notation (dotted lines around the events corresponding to the messages sent). They are indicated on both the send and receive sides to make it clear, though they were not required at the receive side in this case because the messages are received by different agents and therefore necessarily correspond to concurrent events.

Finally the condition `P->size()>1` is reached (and it must be reached because the lifeline of the auctioneer is solid). If it holds (there were more than two participants accepting the bid) the loop continues and a new incremented `cfp` is sent. Eventually, if there are less than two participants bidding, the loop is exited. The auctioneer sends a message to all participants stating the end of the auction and concurrently, if the offered price is higher than the reserved price, the auctioneer sends the message `request` to the participant that offered the price.

It should be clear at this point that we no longer have the problems mentioned in the previous section. Indeed LSCs enriched with our extension offer us the expressive power needed. The example does not illustrate how to describe prohibited behaviour. We show how to specify prohibited behaviour in **Fig. 5**.

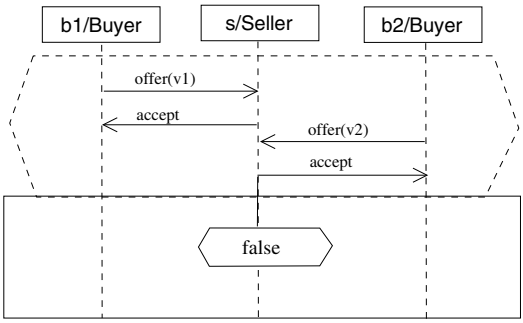


Fig. 5. Prohibiting gazumping.

The LSC in **Fig. 5** illustrates the prohibition of the so-called *gazumping* problem that can happen in the property market in England. A buyer makes an offer to a seller, the seller accepts it and a verbal agreement is made. Later, however, another buyer makes a better offer and the seller decides to go for the better one. Sometimes this happens even without giving the initial buyer the chance to match or increase the offer. Gazumping is by convention not practiced in Scotland. We could therefore imagine the additional norm as described in the LSC above in a property market in Scotland. Agents in Scotland are consequently expected to conform to this norm.

In the prechart of **Fig. 5** we indicate the sequence of communications we want to prohibit: the seller `s` receives an offer from agent `b1`, it accepts the offer. Later `s` receives another offer from another agent `b2` and accepts it as well. Notice that from this point the lifeline of `s` is solid enforcing the instance to progress and enter the body of the chart. The body of the chart contains a hot condition which is false, and when evaluated the chart is aborted. In this way we are describing that the prechart is prohibited.

Notice that modelled this way, an agent *must* conform to the norm. This may be a stronger requirement than intended. Alternatively, we may wish to

allow norms to be violated and impose sanctions on agents that do. This can be modelled by placing the norm in the prechart of an LSC and the corresponding sanction in the body of the chart.

Consider another example with time constraints in **Fig. 6**. This example is taken from [6] and shows how our extended LSCs can be used to model agent contracts and commitments.

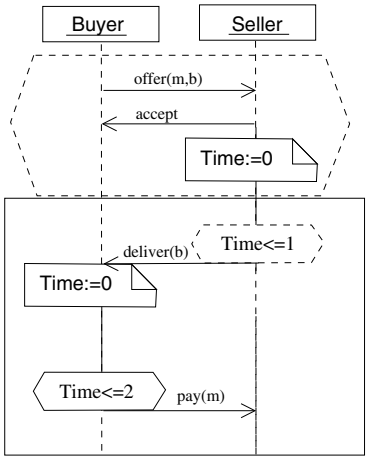


Fig. 6. An agent contract.

This example describes the permitted behaviour of two agents (a buyer and a seller) after they establish an agreement to an exchange of an item for money. According to the contract, the seller has one day to deliver the item to the buyer. If the seller fails to do so the contract is dissolved. After the buyer receives the item it has to pay within two days. At this point we could assume two things: (1) if the buyer fails to pay within two days it violates the contract (2) if the buyer fails to pay within two days there is an incremented fee of £10. We model the first case here. The second case could be modelled easily using a loop similarly to what we have seen in an earlier example.

In the prechart, the buyer offers money `m` in exchange of an item `b` and the seller accepts the offer. The contract is established and the body of the chart is entered. Time is set to 0. The seller is forced to progress along the lifeline, if more than a day passed the condition is false, and because the condition is cold the chart is exited and the contract cancelled. If the seller does deliver the item (the message `deliver` synchronises with the condition and the message is only sent if the condition holds) then time is reset for the buyer. If more than two days have passed for the buyer the condition is false, and because the condition is hot the chart is *aborted* corresponding to a violation of the contract.

We therefore can also use our extended LSCs for describing commitments and contracts between agents.

4 From Agent LSCs to Logic

For space reasons we omit a complete formal semantics of extended LSCs in this paper. LSCs have a formal semantics described thoroughly in other papers, and our extensions should not prove too difficult.

Instead, we briefly hint towards how from an agent LSC we can extract behaviour that can be described in the distributed temporal logic MDTL[11, 10]. In other words, how the specification in an LSC of agent communication, agent commitments, possibly group or society norms and forbidden behaviour is expressed in MDTL. MDTL is, however, not expressive enough to deal with nondeterminism in agent choices and we do not address this issue here. One promising solution to deal with this aspect is given in the Alternating-time Temporal Logic (ATL) [1]. We therefore only concentrate on the mains features of the distributed temporal logic MDTL which concern distribution (locality) and the explicit concurrency operator which can naturally be used to describe concurrent communication between agents.

In the sequel we consider a *generic* LSC an LSC that is defined at the specification level, i.e., that describes an interaction between agent roles rather than agent instances. Let an L be a generic LSC describing an interaction between agent roles. Let R_L be the finite set of roles in L, and A_L be the set of all possible agents in the society that can play one or more roles in R_L. Each role $r \in R_L$ has associated with it a finite set of locations given by $l(r)$. Each location is an intersection point of the role lifeline with a system event. We denote by l_x^r the x^{th} location of role r. A location point can be hot or cold given by a function $temp : l_L \to \{hot, cold\}$ where $l_L = \cup_{r \in R_L} l(r)$.

An LSC event can be either an actual system event of sending or receiving a message, or it could be one of the acts of evaluating a condition, entering a subchart or exiting it. Notice that agent creation, destruction, role changing, and real-time aspects also correspond to events in an agent LSC but we do not consider them here for space reasons. We mainly want to focus on asynchronous inter-agent communication and concurrency. Notice that normally LSCs also allow for communication with the environment, we have not illustrated such aspects here either.

The set of events in L is defined as:

$$E_L = M_L \times \{send, receive\} \cup Cond_L \cup Sub_L \times \{start, end\}$$

where M_L is a set of messages over an alphabet $\mathcal{M}$. Furthermore, we can define a function $event : l_L \to E_L$ that given a location in L returns the associated system event. L induces a partial order on the system events denoted by $\leq_L$.

Events of interest here fall in one of the following categories:

- $(r_1, m!, r_2)$, asynchronous transmission of message m from role r_1 to role r_2
- $(r_1, m?, r_2)$, receipt of message m by role r_2 from role r_1
- $(r_1, a.B\hat{}m!, r_2)$, asynchronous transmission of message m from an agent a playing the role r_1 to a set of agents B playing role r_2

– $(r_1, a.B\hat{}m?, r_2)$, receipt of message m by all agents in B playing the role r_2 from agent a playing role r_1

In the first two cases no specific sender or receiver is given, therefore any agent playing the sender role can send the message and all agents playing the receiver role can receive the message.

A lifeline in an LSC may have a finite set of coregions. Each coregion in a lifeline of a role r denotes a maximal connected set of locations of r. We denote the set of coregions associated to a role r by $cor(l(r))$. If two locations in $l(r)$ given by l^r_x and l^r_y with $x < y$ are within the same coregion $C \in cor(l(r))$ then so are all intermediate locations, i.e., $l^r_z \in C$ with $x \leq z \leq y$. Events in a coregion are unordered (concurrent). For any l^r_x and l^r_y with $x < y$, if there is a coregion C for r with $l^r_x, l^r_y \in C$ then $event(l^r_x)$ *co* $event(l^r_y)$. Otherwise $event(l^r_x) \leq_L event(l^r_y)$.

We now describe briefly the main idea of the distributed temporal logic MDTL. More details can be found in [11, 10]. Each component in a system has a logic to describe internal properties (called *home* logic) and a *communication* logic to describe interactions with other components in the system. A component in this context will be an agent role or a specific agent. The abstract syntax of MDTL_r with r an agent role, is given as follows (in a simplified variant for our purposes):

$$
\begin{array}{l}
\text{MDTL}_r ::= r.H_r \mid r.C_r \\
H_r ::= \text{ATOM}_r \mid \neg H_r \mid H_r \Rightarrow H_r \mid H_r \; \mathcal{U}_\forall \; H_r \mid H_r \; \mathcal{U}_\exists \; H_r \mid \Delta H_r \\
C_r ::= \; Msg_r! \leftrightarrow k.Msg_k? \mid Msg_r? \leftrightarrow k.Msg_k! \mid Msg_r! \rightarrow k.Msg_k? \mid \\
\qquad Msg_r? \leftarrow k.Msg_k! \\
\text{ATOM}_r ::= true \mid T_{\Sigma_s}(X) \; \theta \; T_{\Sigma_s}(X) \mid Msg_r! \mid Msg_r?
\end{array}
$$

The home logic H_r is basically an extension of CTL (notice that $\mathcal{U}_\forall$ corresponds to all paths, whilst $\mathcal{U}_\exists$ corresponds to there is a path) with a concurrency operator Δ. The intuition of a formula $a.(\varphi_1 \wedge \Delta\varphi_2)$ is that from the point of view of a if φ_1 holds then φ_2 holds concurrently. Msg denotes a message term where $Msg_r!$ is used to denote the sending of a message and $Msg_r?$ to denote the receipt of a message. In the communication logic C_r, $\leftrightarrow$ is used for synchronous communication and $\rightarrow$ and $\leftarrow$ for asynchronous communication (send and receive respectively). $T_{\Sigma_s}(X)$ is used to denote data terms with X a set of variables and where θ is a comparison operator. We omit further details here.

When used for describing an LSC, the idea is the following. If we are describing a property of the LSC which only concerns a role (for example that an agent playing a particular role must progress along the lifeline) then it refers to a local property and we use the home logic associated to the role. If by contrast we are describing communication between roles we use the corresponding communication logics of the involved roles. Even though we are not talking about satisfiability here, the idea is that a formula holds at a particular event.

Consider the following example. Let l_1 and l_2 be locations in the lifeline of role r_1, and $l_1, l_2 \in C$ with C a coregion of r_1. Let the events associated to the locations be given by $(r_1, a.b\hat{}m_1!, r_2)$ and $(r_1, a.c\hat{}m_2!, r_2)$. This means that a is sending concurrently a message m_1 to b and a message m_2 to c. In MDTL

the following formula describes what agent a knows at location l_1 (or instead of agent a knows, we can just say that the following formulae has to hold at the event associated to the location):

$$a.(b\hat{}m_1! \wedge \Delta c\hat{}m_2!)$$

that it sends b a message m_1 and concurrently c a message m_2.

If the message m_1 is a hot then it has to be received and consequently the following formula expresses what agent a knows at location l_1 (that eventually the message will arrive):

$$a.(b\hat{}m_1! \Rightarrow F\ b\hat{}m_1?)$$

In the communication logic of a we can also express that a message m_1 is sent to agent b and that b will receive m_1 (because m_1 is assumed to be hot):

$$a.(b\hat{}m_1! \rightarrow b.m_1?)$$

The logic MDTL was developed to specify large distributed object systems [11, 10] and has been used to formalise contracts in component-based design [12]. There is, however, nothing particularly object-oriented in this logic. The distributed logic MDTL and other distributed logics of which MDTL is an extension, namely DTL (sometimes called in distinguished form D_0 and D_1) [9, 8], have been used for describing (temporal) properties of object systems, inter-object communication, object and component contracts. These logics are equally suited for capturing temporal aspects of agents and agent societies. Future work should analyse an extension to MDTL to consider capabilities to distinguish between different sources of nondeterminism in agent systems. ATL [1] is a promising solution with this respect.

5 Discussion

In this paper, we have taken a macro view on agent-based systems and considered a visual notation for modelling agent interaction protocols. We have described some of the extended features of sequence diagrams in AUML and pointed out some of the weaknesses. Moreover, we have shown how LSCs, a very expressive formalism normally used for modelling interactions in object-oriented systems, can also be adapted for agents. Our proposed extension of LSCs for agents is very useful when modelling agent-based systems as it can capture agent interactions, agent commitments and contracts, as well as society norms and forbidden behaviour.

Sequence diagrams and LSCs are both visual scenario-based languages which are appealing to software designers because they convey a simple idea and are intuitive. A further strength of LSCs is that they have a formal semantics. A fully worked out semantics is needed for automatic synthesis and consistency checking. The ability to automatically construct a behavioural equivalent state-based specification (for example statecharts) from the scenarios is a necessary first step towards an ambitious aim of moving automatically to implementation.

Even though there is considerable work on synthesis and consistency checking for LSCs [16, 19] this work needs to be adapted for our agent extension to LSCs. In this paper, we have not given a complete description of the semantics of our extension to LSCs which is currently under development. We have only hinted towards some of the details, and how from parts of an LSC we can derive formulae in a true-concurrent branching time logic MDTL [11, 10]. In MDTL an agent[3] has associated to it two sublogics: a *communication* logic and a *home* logic. An interesting aspect of MDTL and its sublogics is that each corresponds naturally to standard modelling representations in object-oriented design. For instance, the communication logic describes inter-object communication which in UML are given by sequence diagrams, and the home logic describes intra-object behaviour which in UML are given as state diagrams. We also expect that the connection between these sublogics will facilitate synthesis concerns. Future work includes the integration of MDTL into the Edinburgh Concurrency Workbench [23]. We also intend to establish a connection between the workbench and UML 2.0, enabling the verification of temporal formulae over behavioural models.

Further, a main feature of MDTL is the explicit concurrency operator which can naturally be used to describe concurrent communication between agents. However, agent LSCs cannot be fully embedded into MDTL. There are some subtle differences between the properties that we can express with the logic MDTL and with an agent LSC. This difference concerns the nondeterminism in agent choices. A promising solution lies in the Alternating-Time Temporal Logic (ATL) of [1].

Finally, this work can also be seen as a providing a notation for engineers which is compatible with the formal framework of [6, 7]. The logic LCR given in [6] is an extension of CTL* with deontic modalities which enables the specification of social norms and interaction contracts. As we have shown, agent LSCs are useful for capturing these aspects as well. The expressiveness of both formalisms is not quite comparable though. Agent LSCs can express concurrency and nondeterminism in agent behaviour, which LCR cannot. Nonetheless, it would be interesting to investigate whether an agent LSC can be embedded in a powerful combined logic consisting of the concurrent features of MDTL with deontic aspects of LCR and ATL's capabilities to distinguish between different sources of nondeterminism.

References

1. R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time Temporal Logic. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, pages 100–109. IEEE Computer Society Press, 1997.
2. B. Bauer, J. Müller, and J. Odell. Agent UML: A formalism for specifying multiagent interaction. In P. Ciancarini and M. Wooldridge, editors, *Agent-Oriented Software Engineering*, volume 1957 of *Lecture Notes in AI*, pages 91–103. Springer, 2001.

[3] Or equally an object or component, which in MDTL is simply referred to as *module* hence the abbreviation - Module Distributed Temporal Logic.

3. B. Bauer, J. Müller, and J. Odell. Agent UML: A formalism for specifying multiagent software systems. *International Journal of Software Engineering and Knowledge Engineering*, 11(3):207–230, June 2001.
4. W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.
5. R. Depke, R. Heckel, and J. Küster. Roles in agent-oriented modeling. *International Journal of Software Engineering and Knowledge Engineering*, 11(3):281–302, June 2001.
6. V. Dignum, J.-J. Meyer, F. Dignum, and H. Weigand. Formal specification of interaction in agent societies. In *2nd Goddard Workshop on Formal Approaches to Agent-Based Systems (FAABS), Maryland, Oct, (2002)*, 2002.
7. V. Dignum, J.-J. Meyer, H. Weigand, and F. Dignum. An organization-oriented model for agent societies. In *Proceedings of International Workshop on Regulated Agent-Based Social Systems: Theories and Applications (RASTA'02), at AAMAS, Bologna, Italy, 16 July 2002*, 2002.
8. H.-D. Ehrich and C. Caleiro. Specifying communication in distributed information systems. *Acta Informatica*, 36(Fasc. 8):591–616, 2000.
9. H.-D. Ehrich, C. Caleiro, A Sernadas, and G. Denker. Logics for Specifying Concurrent Information Systems. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, pages 167–198. Kluwer Academic Publishers, 1998.
10. J. Küster Filipe. *Foundations of a Module Concept for Distributed Object Systems*. PhD thesis, Technische Universität Braunschweig, Germany, September 2000.
11. J. Küster Filipe. Fundamentals of a module logic for distributed object systems. *Journal of Functional and Logic Programming*, 2000(3), March 2000.
12. J. Küster Filipe. A logic-based formalization for component specification. *Journal of Object Technology*, 1(3):231–248, 2002.
13. S. Flake, C. Geiger, and J. Küster. Towards UML-based analysis and design of multi-agent systems. In *International Symposium on Information Science Innovations in Engineering of Natural and Artificial Intelligent Systems (ENAIS'2001), Dubai, March 2001*, 2001.
14. FIPA (Foundation for Intelligent Physical Agents). *FIPA English Auction Interaction Protocol Specification*. available from www.fipa.org, August 2001.
15. FIPA (Foundation for Intelligent Physical Agents). *FIPA Modelling: Interaction Diagrams, Working Draft*. FIPA version 2003-07-02, available from www.fipa.org, July 2003.
16. D. Harel and H. Kugler. Synthesizing state-based object systems from LSC specifications. *International Journal of Foundations of Computer Science*, 13(1):5–51, 2002.
17. D. Harel and R. Marelly. Playing with Time: On the Specification and Execution of Time-Enriched LSCs. In *10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'02) October 11 - 16, 2002, Fort Worth, Texas*, pages 193–202, 2002.
18. D. Harel and R. Marelly. *Come, Let's Play: Scenario-based Programming Using LSCs and the Play-Engine*. Springer, 2003.
19. D. Harel and R. Marelly. Specifying and executing behavioural requirements: the play-in/play-out approach. *Software and System Modeling (SoSyM)*, 2:82–107, 2003.

20. M.-P. Huget. Extending Agent UML Sequence Diagrams. In F. Giunchiglia, J. Odell, and G. Weiß, editors, *Agent-Oriented Software Engineering III: Third International Workshop, AOSE 2002, Bologna, Italy, July 15, 2002*, volume 2585 of *LNCS*, pages 150–161. Springer, 2003.
21. J. Klose and H. Wittke. An automata-based interpretation of live sequence charts. In T. Margaria and W. Yi, editors, *TACAS 2001*, volume 2031 of *LNCS*. Springer, 2001.
22. R. Marelly, D. Harel, and H. Kugler. Multiple instances and symbolic variables in executable sequence charts. In *Proc. 17th Ann. AM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '2002), November*, pages 83–100, 2002.
23. F. Moller and P. Stevens. Edinburgh Concurrency Workbench user manual (version 7.1). Available from http://www.dcs.ed.ac.uk/home/cwb/.
24. J. Odell. Objects and agents compared. *Journal of Object Technology*, 1(1):41–53, May-June 2002.
25. J. Odell, H. Van Dyke Parunak, and B. Bauer. Representing agent interaction protocols in UML. In P. Ciancarini and M. Wooldridge, editors, *Agent-oriented Software Engineering*, volume 1957 of *Lecture Notes in AI*, pages 121–140. Springer, 2001.
26. J. Odell, H. Van Dyke Parunak, S. Brueckner, and J. Sauter. Temporal aspects of dynamic role assignment. In *Proceedings of the Fourth International Workshop on Agent-Oriented Software Engineering (AOSE-2003)*, 2003.
27. OMG. *UML 2.0 OCL Specification, Version 1.6*. OMG document ad/03-01-07, available from www.uml.org, August 2003.
28. OMG. *UML 2.0 Superstructure Draft Adopted Specification*. OMG document ptc/03-08-02, available from www.uml.org, August 2003.
29. OMG. *Unified Modelling Language Specification, version 1.5*. OMG document formal/03-03-01, available from www.uml.org, March 2003.
30. M. Wooldridge, N. R. Jennings, and D. Kinny. The Gaia methodology for agent-oriented analysis and design. *Journal of Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.
31. Z.120 ITU-TS Recommendation Z.120. *Message Sequence Chart (MSC)*. ITU-TS, Geneva, 1996.

Organising Computation through Dynamic Grouping

Michael Fisher[1], Chiara Ghidini[2,⋆], and Benjamin Hirsch[1]

[1] Department of Computer Science, University of Liverpool, Liverpool L69 7ZF, UK
{M.Fisher,B.Hirsch}@csc.liv.ac.uk
[2] Automated Reasoning Systems Division (SRA), ITC-IRST, Trento, Italy
ghidini@itc.it

Abstract. There are a range of abstractions used for both modelling and programming of modern computational systems. While these abstractions may have been devised for specific purposes, the variety of options is often confusing, with development and programming techniques often being distinct. The aim of this chapter is two-fold. First, we bring together a number of these abstractions into one, general, view. Second, we show how, by grouping computational elements, this general view can capture a range of behaviours in areas such as multi-agent systems, web services, and object-oriented systems. This framework then provides a basis for design and implementation techniques for a wide variety of modern computational systems, in particular providing the basis of a general programming language for dynamic, distributed computation.

1 Introduction

In the past there were just *programs* and *procedures*. Then *processes* and *objects*. Now, we also have *components*, *agents*, *software features* and *web services*. While these abstractions are used in a variety of different application areas, there is a great deal of similarity between these concepts. All concern independent, typically concurrent, entities whose basic dynamic behaviour is modified by the specific contexts in which they are used. Thus,

- the behaviour of an object-oriented component is modified by its position within an object hierarchy, typically its relationship to other components represented via inheritance,
- the behaviour of an agent is modified by its own internal goals, together with its views of other agents,
- the behaviour of a feature is modified by the other features present, as each feature only represents a partial view of the computational element, while
- the behaviour of a web service is modified by the ontology it uses, together with the web context within which it is invoked.

And so on.

The range of abstractions used in both modelling and programming is often confusing. When designing modern software, are we concerned with objects or agents? Should

⋆ Work supported by MIUR under FIRB-RBNE0195K5 contract.

M.D. Ryan et al. (Eds.): Objects, Agents, and Features, LNCS 2975, pp. 117–136, 2004.

these distributed elements be seen as services or agents? Should they be components or objects? And so on. In an effort to simplify this situation, we here bring together a number of these abstractions into one, general, view. Later, we will show how, by grouping computational elements, this general view can capture a range of behaviours in the above areas. This framework then provides a basis for design and implementation techniques for a wide variety of modern computational systems, in particular providing the basis of a general programming language for dynamic, and potentially distributed, computation.

1.1 Concepts

While there is often little agreement on the definitions of the elements we consider in this chapter, we will use the following working descriptions.

Object: An object is a computational entity encapsulating both data and behaviour [15]. Objects react to messages received by invoking *methods*. In object-based systems where inheritance is present, messages that can not be dealt with by an object may be passed on to other objects using inheritance. Often, but not always [12], objects are clustered into *classes* in order to capture common functionality/behaviour [7].

Agent: An agent is an *autonomous* computational entity encapsulating data and behaviour [17]. In contrast to objects, agents have control over how they react to their environment, and may adapt their internal behaviour to evolving situations [6]. Agents often work in unpredictable environments and so may have only subjective representations of aspects outside their control [16].

Software Component: A component is a software object encapsulating specific functionality and interacting with other components through its defined interface. A component has a clearly defined interface and conforms to a prescribed behaviour common to all components within the software architecture [11, 9].

Feature: A feature is an optional unit of functionality that may be added to, or excluded from, a system [4]. A feature is usually perceived by the user as having a self-contained functional role. Features may override the default behaviour of the system, or introduce a new behaviour. As a consequence they may interact or interfere in unexpected ways and thus cause the overall system behaviour to be undesirable [10].

Web Service: A software system identified by a URI, whose public interfaces and bindings are defined, and described, using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols [13]. This framework provides the basis for ambitious projects such as the Semantic Web [1, 14].

1.2 Grouping

In this chapter we will primarily consider the *organisation* of computation within *collections* of these entities. In particular, we show how a simple framework of computation within groups can provide a basis for a range of activities of contemporary software artifacts such as those above. Thus, grouping together each of these types of artifact give us a range of different possible behaviours for each type:

Objects. The grouping of objects together is widespread in class-based systems as classes can be seen as groups of objects sharing common properties. Similarly, inheritance can be seen as the linking of groups of objects. More interestingly, grouping can be seen as a practical mechanism for both structuring the object space and implementing forms of communication and fault-tolerance [3, 2].

Agents. It is common, within agent-based systems, to deal with large, dynamic, groups of agents. In such multi-agent systems, sub-grouping is not only used to model both cooperative and competitive structures, but can also be used to organise the agents into structures corresponding to physical situations (e.g. all agents in the same vicinity are grouped together). In all these cases, it is natural to have the notion of dynamic movement of agents between groups and the dynamic creation and destruction of groups [17].

Components. In component-based software engineering [11], it is the construction of the group of components, rather than the construction of single components, that is important. A collection of components forms a *component architecture*, and each component architecture defines the conventions by which components are linked together. Collecting together suitable components and ensuring they interact in an appropriate way is the key mechanism, though such collection is usually carried out by the programmer/user [9].

Features. A increasingly common way of designing complex systems consists in thinking of the system as being composed of a basic part plus a collection of features that can be incrementally added to the system to provide the required services. In such systems, it is the appropriate integration of features that is crucial, as features may interact with each other in ways that are difficult to anticipate, and may cause the overall system behaviour to be undesirable [10].

Services. The idea behind web services is that appropriate services can again be collected together in order to provide an application working in a distributed way over the WWW. The difference between this approach and that of many (but not all) of the others above is that the services are typically meant to be found and grouped automatically. This search for appropriate services is supported by the the idea, from the *Semantic Web*, of publishing a service's properties and then letting other agents/applets browse/select these published specifications in order to choose the most appropriate service [14, 1].

1.3 Structure of the Chapter

The structure of this chapter is as follows. Our general approach to the organisation of entities in collections (or groups) is outlined in Section 2, where we present some simple examples of activities involving collections of entities. Our goal is to provide a framework where computation is organised within collections of entities provided by groups. Thus, Section 3 presents the implementation of the grouping mechanism presented in Section 2 together with further examples. Section 4 contains higher-level views of how to use our framework in a variety of application areas and, finally, Section 5 provides some concluding remarks.

2 Elements and Groups

In order to avoid calling the basic elements we deal with by some name that already has numerous definitions, e.g. 'objects' or 'agents', we will simply call them 'elements'.

2.1 Elements

Elements are the basic entities within our model. Elements:

- are opaque, encapsulating both data and behaviour;
- are concurrently active, in particular they are asynchronously executing; and
- communicate via message-passing.

Specifically, the basic communication mechanism we provide between elements is *broadcast* message-passing. Thus, when an element sends a message it does not necessarily send it to a specified *destination*, it merely sends it to its environment, where it can be received by *all* other elements within the environment. Although broadcast is the basic mechanism, both multi-cast and point-to-point message-passing can be implemented on top of this.

The default behaviour for a message is that if it is broadcast, then it will *eventually* be received at all possible receivers. Note that, by default, the order of messages is not preserved, though such a constraint can be added if necessary. Individual elements only act upon certain identified messages. Thus, an element is able to filter out messages that it wishes to recognise, ignoring all others.

2.2 Groups

A group contains a set of elements. Groups understand certain messages, in particular those relating to group manipulation, such as adding, deleting, etc. Thus, the basic properties we require of groups are that elements should be able to

- send a message within a group,
- ascertain whether a certain element is a member of a group,
- add an element to a group,
- remove a specified element from a group, and,
- construct a new subgroup.

There are other properties that might be useful, such as the ability to list the members of a group, but they are not essential.

Groups contain (references to) other elements which may, in turn, be either simple elements or groups. Groups may also overlap. Thus, one element may be a member of several groups. When an element within a group sends a message, the default behaviour is to pass the message on to all the members of the groups.

While the basic purpose of groups is to restrict the set of receivers for broadcast messages, and thus provide finer structure within the element space, more complex applications of groups can be envisaged. For example, the group might enforce a different internal model of communication [5]. Thus, groups can effectively enforce their own 'meta-level' constraints on message-passing between elements, for example specifying that the order of messages is preserved.

2.3 Element ≡ Group

To simplify and clarify the model, we actually identify the notions of element and group. Thus, groups are also elements. More surprisingly, perhaps, elements are also groups. Consequently, basic elements are groups with no contents. One of the values of identifying these two concepts is that any message that can be sent to an element, can be sent to a group (concerning cloning, termination, activation, etc) and vice versa (concerning group membership, addition to groups, etc). Thus, not only can elements be members of one or more groups, but they can also serve as groups for other elements. Further advantages of this approach are:

- communication (which, in our approach, is based on broadcasting) can be limited to particular networks of groups;
- since elements have behaviour, and groups are also elements, groups can also have internal policies and rules. Therefore groups are much more powerful than mere "containers"[1];
- the inherent structure can be exploited to group elements based on different aspects such as problem structure, abilities, and even meta-information such as owner, physical location etc; and
- groups can capture key computational information, e.g. groups can be transient, can be passed between elements, and can evolve, while long-lived groups may also capture the idea of "patterns of activity".

Thus, from the different behaviours of elements, different group behaviours can follow. And vice versa.

Finally, it is important to note that, while groups are initially designed/programmed with a specific behaviour (e.g. controlling entry to the group, controlling communication within/into the group, or controlling how information is shared within the group), in principle they can 'learn'/adopt new behaviour as computation proceeds. Thus, the group of elements can evolve, both in terms of content and in terms of behaviour, over time.

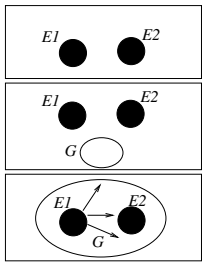
[1] As an example, in [8] we showed how simple rules could be used to formulate different behaviours within agents, and how those behaviours influence the resulting group structures.

2.4 Simple Examples

We consider here three simple, yet very common, examples of behaviour involving elements belonging to some sort of collections, and we illustrate how to represent these behaviours in our framework.

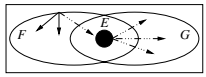
Example 1 (Point-to-Point Communication)
In order to provide a mechanism for element E1 to communicate directly to (and only with) element E2, E1 may

1. *make a new element, called G, and add itself to G,*
2. *broadcast a message suggesting E2 joins G, to G,*
3. *then broadcast a message, m, within G.*

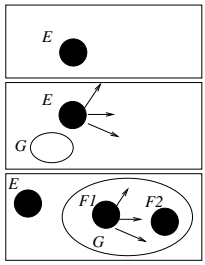


Note that the decision about whether E2 actually joins G is likely to be up to E2 itself. Similarly, we are not considering security aspects here, for example ensuring no other elements, for example E3, join G.

Example 2 (Filtering)
Element E is a member of both F and G. When F broadcasts messages to its contents, E will receive these and can pass on selected messages to G. Thus, E acts as an intermediary between F and G.

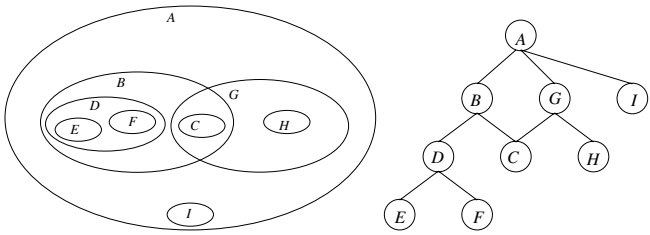


Example 3 (Interest Groups)
Element E wishes to collect together all those elements that have interest in, or capability to solve, a certain problem. Thus, E creates a new element, G. E then broadcasts a message, with G as one of its arguments, asking for interested parties to join G. (Note that the message has sufficient information encoded within it for any receiving element to be able to work out what capabilities are being requested.) When such an element, say F, receives this message, it can join G. Thus, G contains all those elements who have joined based on this message and messages broadcast throughout G will just be received by those elements.



3 Implementation

We have implemented the general grouping approach, as outlined above. In order to provide the basis for as wide a range of applications as possible, this implementation is based on Java. Thus, elements are essentially Java threads, with additional classes for organising group structures and communication.



Element	CN	CX
A	{B,G}	∅
B	{D,C}	{A}
C	∅	{B,G}
D	{E,F}	{B}
E	∅	{D}
F	∅	{D}
G	{A}	{C,H}
H	∅	{G}
I	∅	{A}

Fig. 1. Different views of the system structure

3.1 Implementing Groups

As mentioned before, we identify the notions of elements and groups. So, every element has the potential to contain other elements. What we consider to be "basic elements" are essentially elements with empty contents. This organisation is reflected in each element by using three sets: *Content*, *Context* and *Known*. Thus, each element has

- a *Content* set, describing the set of elements that are *members* of the element (group),
- a *Context* set, describing the set of groups (elements) that the element is a *member of*, and
- a *Known* set, describing the set of *known* elements, typically outside the content/-context hierarchy.

Fig. 1 illustrates a typical group structure of our system, the corresponding hierarchical structure, and the representation of this structure using the *Content*/*Context* (**CN**/**CX**) mechanism. Note that we will use different representations throughout the chapter, as they are more or less suited to different situations.

While the pure *Content*/*Context* approach (that is, without the third set *Known*) is simple and intuitive, it has problems. In order for an element to send a message to another, far-away, element, it has to know the topology of the element space in order to avoid broadcasting messages to the whole system. Thus, each element also keeps a list of *known* elements, with which it can communicate directly. When an element encounters a previously unknown element, it adds it to the *Known* set. This allows efficient communication with elements outside the current hierarchy/structure. It should be stressed at this point that we include the *Known* set for convenience and efficiency, rather than pressing theoretical matters. We chose to implement a third set of direct connections as we aim to build a practical tool, and wish to reduce the broadcasting of messages to the entire system when elements have to communicate with other, far-away, elements. Further, it may be difficult to see how such a *Known* set may be established in the first place. With regard to this there are two points to note:

1. since the group structures are dynamic, elements which reside in a common *Content*/*Context* hierarchy at one point may not be so at a later moment in time — thus, retaining references to relevant elements may be useful;

2. even if two elements are in the same *Content*/*Context* hierarchy, it may be more efficient for one to store a direct reference to to the other than for general communication to be used.

Since the topology of the element space is dynamic, it can adapt to accommodate new structures to solve problems. This is typically achieved by two general methods. First, an element can move through the element space and join a group that can help it solve its problem (or alternatively it can invite other elements to join its content). Note that "movement" does not necessarily mean that elements leave one group to join the other — they can join many groups and have many elements in their content. Also, movement refers to virtual, rather than physical, movement. Due to the ability to send messages to sub-groups, elements can restrict the amount of superfluous messages further. Alternatively, elements can create "dedicated" group elements that serve as containers for elements that share some property (typically, a common ability).

Whether elements are invited to *Content*, *Context*, or to dedicated groups, the duration of their stay can be either transient (that is, they do what they are asked and then leave), or (semi) permanent (that is, elements stay in the group even after they have completed their task). Duration of stay depends on both the element and its context.

3.2 Implementing Communication

As described above, elements are implemented using threads. Attached to each element is a *message buffer*, effectively providing the *inbox* for that element. Thus, messages, which are instances of a specific `Message` class, are added to an element's message buffer in order to implement communication. Typically, elements read all messages received in a given cycle and act upon them. Naturally, they can also send messages and/or carry out computations during a cycle.

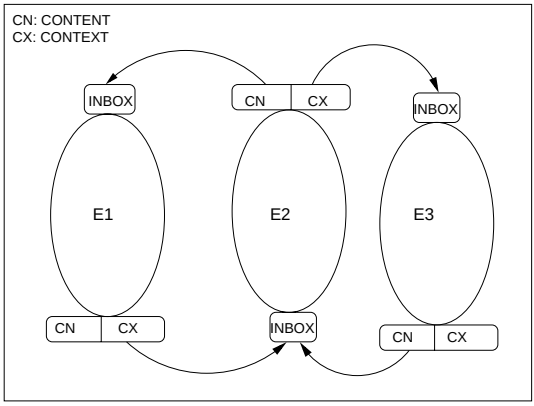


Fig. 2. Implementation of communication

Fig. 2 illustrates how we implement communication by providing a particular example. Each element has an *inbox* to which others can link, either through their *Content*, or

Context (or *Known*, which we do not show in the picture). Generally if an element connects to another one through it's *Content*, there will be a reciprocal connection through the latter's *Context*, and vice-versa, in order to comply with the grouping theory.

While the basic requirement for communication is to be able to send messages to elements in the *Context*, *Content*, or *Known* sets, we often require more sophisticated message-passing behaviour. Thus, instead of sending messages just to *Content*, *Context*, or *Known*, an element can specify messages to be sent to an arbitrary subset of elements from these sets. This is achieved by the statement

```
send(Target,Msg)
```

where `Target` is a target set and `Msg` is the instance of `Message` being sent. Target sets can be specified using the set operations *union* (S_1 `union` S_2), *intersection* (S_1 `intersection` S_2), and *exclusion* (S_1 `without` S_2), where S_1 and S_2 can themselves be single elements, sets of elements, or set expressions[2]. Thus,

```
send(Content without i, message)
```

sends `message` to all members of the element's *Content* set *except* element `i`.

Additionally, messages can be nested. Thus, if an element receives a `send` message, it will interpret the `target` relative to its own *Content*/*Context*/*Known* sets. For example, element `i` can send the following message

```
send(Context, send(Content without i, message))    (1)
```

Then, all elements in `i`'s *Context* receive the message

```
send(Content without i,message)
```

and re-send `message` to all their *Content*, except for element `i` (which originally sent the message). So, using (1), `i` can send messages to elements that are "on the same level" as itself, i.e., that share at least one of the same *Context* elements.

Since nesting of message sends is so common, we also have a special send command `sendAll`, which is interpreted by elements to recursively re-send the message to the same set (relative to the receiving element). For example,

```
sendAll(Content, m)
```

will ensure that message `m` will be recursively sent to all members of *Content* sets 'below' the sending element. Also,

```
sendAll(Content union Context, m)
```

will ensure that `m` is propagated to the whole element space. The implementation of `sendAll` is based on the equivalence (when expanded within element `i`)

$$\texttt{sendAll}_{\texttt{i}}\texttt{(Set,Msg)} \equiv \texttt{send(Set,Msg)} \wedge \texttt{send(Set,sendAll(Set\i,Msg))} \quad (2)$$

[2] Note that we sometimes use the mathematical notations $\cup, \cap$, and $\setminus$ for the set operations `union`, `intersection`, and `without` respectively. We also drop subscripts whenever possible to enhance readability.

For example, element `i`, upon receiving `sendAll(Content union Context, m)`, will execute two sends:

`send(Content union Context, m)`, and

```
send(Content union Context,
     sendAll((Content union Context)\i, m)).
```

By removing itself from the list of elements that are to receive the message, the element avoids messages being sent over and over again through the element space.

3.3 Primitive Actions

Before listing some of the primitive actions available to an element, we note that the exact implementation of these actions effectively specifies the level of autonomy the agents have. For example, if the elements have a high level of autonomy, then messages sent to them involving movement are just 'suggestions' — the element still decides itself whether to move. However, if we require little autonomy (e.g. typical within object-based systems), these primitives might be implemented to succeed automatically. And there can be a range of different levels between these two extremes. Thus, in describing the primitive and composite actions in this, and the next, section, we must bear in mind the effect that the level of autonomy required may have.
We begin with simple primitives.

`send(Set,Message)` and `receive` are arguably the most important primitive actions. As stated above, messages are sent to a subset of the union of *Content*, *Context*, and *Known* sets. `send` understands not only basic, but also composite sets, such as those described in Section 3.2. `receive` reads *all* messages that are in the element's message buffer and processes them.

`sendAll(Set,Message)` works in a similar way, with the difference that the message is sent twice, once as normal message, and once wrapped into another `sendAll` message. This message is then interpreted by the receiving elements in order to re-send the original message to their respective sets. Each element adds itself to the exclusion list (as in (2) above) so as to avoid receiving the same `sendAll` message again. This method, while very intuitive, does not completely avoid messages being sent more than once to an element, as the message may be able to reach it through several different routes.

`addToContent(Element)` and `addToContext(Element)` are the primitives that allow elements to move within the element space. The element receiving such a message adds the argument to the appropriate set and replies directly back to the argument element with an `addedToContent` or `addedToContext` message. Upon receiving confirmation, the initiating element moves the message buffer of `Element` from *Known* to the appropriate set. This method reduces messages sent though the element space.

`disconnect` is sent if elements want to remove an element from *Content*, *Context*, or *Known*. If received, the element replies with `disconnected` and removes the relevant

connection. While elements can't refuse to be disconnected, we adopt the request/acknowledge technique in order to give the element that is to be disconnected a chance to send final messages before severing the connection.

`create` allows an element to create a new element. This is mainly intended for creating groups in order to collect elements together[3].

3.4 Composite Actions

With the above actions we can now define several higher level behaviours. Again, they primarily concern the movement within the element space.

`moveUp(Set)/goUp(Set)` allows an element to move up within the hierarchical structure (into the group(s) of its context elements). While `moveUp` is initiated by the element that moves, `goUp` would be sent to an element by one of the elements in its *Context*. Fig. 3 illustrates the difference between the two composite actions. The figures on the top represent the group structure before the composite actions are executed, while the bottom part represents the new group structures after the execution of the composite actions.

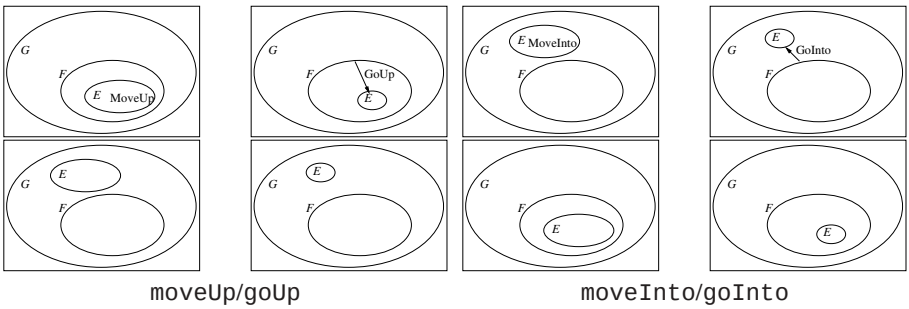


Fig. 3. The composite actions `moveUp`/`goUp` and `moveInto`/`goInto`

`moveInto(Set)/goInto(Set)` is the reverse of the up-movement, as can be seen in Fig. 3. An element moves into one (or more) of its *Content* elements. Again, `moveInto` is initiated by the moving element, while `goInto` is initiated by the one that will receive the moving element.

The `die` predicate does what it says — it makes an element stop its computation gracefully. After receiving such a message, the element sends `addToContent` or `addToContext` messages for each element in the respective sets to *Content* respectively *Context*. Elements in *Context* can add the content elements to their content and

[3] At present, only hard-coded elements can be created, but there is infrastructure in place to allow elements to determine what messages the created element should understand, and how to react to them.

vice versa. It then sends *disconnect* messages to all elements, and stops the thread[4]. Note that in order to just remove an element, one can simply send a disconnect message to *Content* and *Context*, and stop the thread.

`merge(Element)` allows elements to inherit their *Content* and *Context*, such that all *Context* elements of the one being assimilated will be in the *Context* of the assimilating elements, and ditto with *Content*. (Note that there are a number of issues concerning this operation that we will not consider here, for example how to combine the behaviours of elements as well as the contents.)

`clone` makes a copy of an element. As an element is not only defined by its internal programming but, to a substantial part, by its connections with other elements, a "real" clone needs to negotiate connections with those. (Note that we cannot guarantee a clone operation will succeed, as elements might refuse to allow cloned connections.) While the clone operation does attempt to re-connect, we must recognise that this might not always be possible.

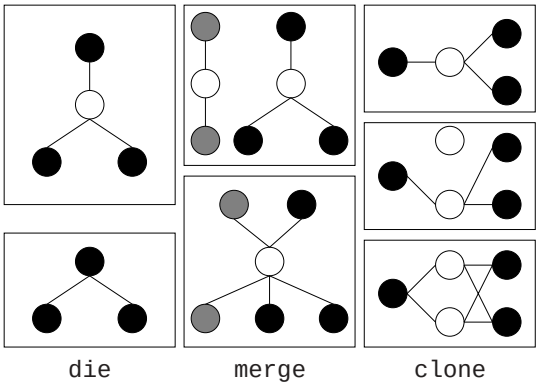


Fig. 4. The composite actions `die`, `merge`, and `clone`

Fig. 4 gives a graphical representation of the effect of the composite actions `die`, `merge`, and `clone` on the structure of the system. The composite actions are initiated by the element(s) represented as an empty circle.

3.5 Messages

In our framework, messages are instances of class `Message`. Each message contains a message name (predicate name) and an arbitrary number of arguments. For ease of use, we refer to arguments using strings, rather than their position (as is common).

[4] The reasoning behind connecting the *Content* and *Context* elements is to avoid disconnected (sets of) elements. While it is possible to re-connect using the underlying *requestChannel* method, it would require elements to know at least one element by name. At this stage, we assume that all elements are connected in one graph.

```
//  createP2P takes the name of the group to be
//  created, as well as the element we wish to talk to,
//  as arguments, and returns the new group element

Element createP2P(String elementName,Element e)
{
  Element p2p = new GroupElement(elementName);
  p2p.addToContent(this);
  p2p.addToContent(e);
  new Thread(p2p).start();
  return p2p;
}

//  ... create the group:

Element G = createP2P("G", E2);

//  ... have something to say:

Message m = new Message(this, "send");
m.addArg("message",message);

//  ... have some set to say it to:

m.addArg("set",new SetExpression("content"));

// ... and say it!
send(new SetExpression("element", "G"),m);
```

Fig. 5. Code fragment defining group supporting point-to-point communication

Arguments can be either part of the actual message content, or meta-data such as sender, messageID, timestamp, etc.

3.6 Code for Simple Examples

This section attempts to provide the reader with a flavour of what the actual code for programming elements looks like. We show how to implement the examples outlined in Section 2.4. While these might be very simple examples, they provide an indication of the ease with which we can cater for different requirements. For readability we omit details such as exception handling, as it adds neither to readability nor understanding of the code.

Example 4 (Point-to-Point Communication) *In Fig. 5, a simple mechanism for point-to-point communication is implemented. The method* `createP2P(String, Element)` *creates a new group element (based on the class* `GroupElement` *which basically just forwards messages) containing itself and the element it wishes to communicate with.*

Example 5 (Filtering) *This example, given in Fig. 6, shows the method* `parseMessage(Message)`*, a method which is called for each message that is read at one cycle. The super class provides standard behaviour for the above mentioned basic and composite actions. Generally, at least a subset of those will be adapted by elements. The code fragment shows how elements can act as intermediaries between different groups (elements). Here, the actual decision on whether or not to forward a message is carried out by the method* `isValid(Message)`*. If the message should be forwarded, the element creates a new message* `send(Content,Message)`*, which is sent to group G.*

```
 void parseMessage(Message m)
{
   // check whether m is to be forwarded

   if (isValid(m))
  {
     // yes, so send m to content of receiving group
     Message new_message = new Message(this, "send");
     new_message.addArg("set",
                        new SetExpression("content"));
     new_message.addArg("message", m);
     send(new SetExpression("element","G"),new_message);
  }
   else   // if not, we do whatever we should be doing
     super.parseMessage(m);
}
```

Fig. 6. Code fragment defining message filtering behaviour

Example 6 (Interest Groups) *In this example, presented in Fig. 7, we show how an element creates an interest group (in this case it looks for other elements that are able to add two numbers), and broadcasts an invitation to the element space.*

4 Applications

While we considered detailed implementation of grouping structures in the last section, we here take a broader view of the form of groups we are developing. In particular, we consider, at a high-level, how such grouping might be used in objects, agents, features and services.

4.1 Objects: Sharing Using Classes and Inheritance

As an exercise, we can use the group structuring mechanism to simulate an object-oriented class hierarchy. Generally, object oriented programming languages offer at least some of the following features [3]: encapsulation; inheritance; polymorphism; abstraction.

```
// Create the group element
Element group = new GroupElement("group");
System.out.println(this+" Created group" group);
new Thread(group).start();
System.out.println(this+" Started group element");

// Add new group to creating element's context
addToContext(group);

// Create message representing ability required
Message needed = new Message();
needed.setContent("do");
needed.addArg("ability","sum");
needed.addArg("arg1",new Integer(2));
needed.addArg("arg2",new Integer(3));
Message a = new Message(this,
                    "neededAbility",
                    new NamedList("needed",needed));

// Send message to initialise group element
send(new SetExpression("element",group),a);

// Create message to find element able to
// undertake the requested ability
Message question = new Message("isAble");
question.addArg("ability","sum");
question.addArg("arg1",new Integer(2));
question.addArg("arg2",new Integer(3));
question.addArg("goTo", group);

// Send 'finding' message to all other elements
sendAll(new SetExpression("without",
       new SetExpression("union","content","context"),
       new SetExpression("element",group)),question);
```

Fig. 7. Code fragment creating an interest group

Encapsulation we get for free, as elements by their very nature encapsulate both data and behaviour.

Inheritance can be simulated in a straightforward manner. *Context* and *Content* contains a link to the "super-element" and children respectively. Now, whenever an element encounters a method it does not implement, it sends a request to its *Context*, which in turn either sends back the computed value, or again forwards the query to its super-element. Note also that we natively can support multiple inheritance, if desired, by allowing more than one element to be in the *Context*.

Polymorphism entails that objects can override methods they would otherwise inherit from their super-class. This again is straightforward in our system, as we only need to provide the element with the relevant methods.

Abstraction, i.e. abstract methods, as well as classes, can also easily be integrated. Recall that joining groups works by sending a request to the group element, which then can decide whether or not to allow the asking element to join. Groups can simply only allow elements into their *Content* that adhere to certain rules — in this case they would need to provide a certain method, or a set of methods.

4.2 Objects: Sharing through Groups

In the majority of object-based systems, object behaviour is shared by using *inheritance*. However, the utility of inheritance relies on the assumption that the inherited behaviour is 'good', and is delivered reliably. We believe the first constraint can be seen as violating the basic principle of objects, i.e., that they control their own behaviour[5], while the second does not sit well in either a distributed or concurrent environments where faults may occur.

We can provide an alternative mechanism: if an object can not deal with a message on its own, it broadcasts a request to its *Context* asking how to do something, e.g., implement a particular method. The object collects the replies and *decides for itself* which one to use. The decision/resolution process within the object might be simple, e.g., choose the first reply, or choose the reply sent by the 'most preferred' object, or could be much more complex, e.g. based upon the 'goodness' of the proposed solution! The important fact there is that the object can decide what to do according to the rules that describe their behaviour.

This approach can be seen as a hybrid between objects and agents, with the elements concerned being allowed a small degree of autonomy in choosing the 'best' solution.

4.3 Agents: Cooperation and Competition

By using groups and broadcast message-passing, we are able to represent typical complex interactions among agents, namely cooperation and competition.

Imagine a scenario where agents are competing, for example by *bidding* for resources. In this case, cooperative activity within groups of agents can be organised so that the group as a whole puts together a bid. If successful, the group must (collectively) decide who to distribute the resource to. Thus, a number of groups might be cooperating internally to generate bids, but competing (with other groups) to have their bid accepted.

Within a given group, various subgroups can be formed. For example, if several members of a group are 'unhappy' with another member's behaviour, they might be able to create a new subgroup within the old grouping which excludes the unwanted agent. Note that members of the subgroup can still receive the outer group's communications, while members of the outer one cannot receive the inner group's communications by default (that is unless an explicit message is sent to them by the inner group). Although we have described this as a retributive act, such dynamic restructuring is natural as groups increase in size.

[5] Often in implementations of inheritance, behaviour (typically method bodies) is copied directly into the object from where it is inherited.

Although these examples have been based upon agents competing and cooperating in order to secure a certain resource, many other types of system can be represented. In particular, in defining the appropriate behaviour for the entities in the system we can represent *agent societies*, where elements seen as individuals cooperate with their fellow group members, but where the groups themselves compete for some global resource.

4.4 Feature Interaction

A complex system can be described as some set of basic behaviours that can be extended with extra features. A prominent example is a phone system where features such as "ring back when free" are added. We may use our system to model this as follows. The main group sends all events that can happen (such as placing a call) to its *Content*. In this content, we have elements (intuitively corresponding to individual features) containing rules that describe the actions that need to take place given an event. The basic system just has the base element as *Content* member. We can, however, add feature elements to the *Content*. They also receive the events, and can act accordingly.

While this is straightforward, it does not solve the problems of feature interaction. We can, in principle, utilise techniques that have been (and are being) produced in the field of feature interaction detection and resolution [4]. In particular we may think of applying different approaches to on-line techniques depending on whether the control of the event is located at the group level (thus using the group as a feature manager), or at the individual feature level (thus allowing features to engage in negotiation processes to find an acceptable solution). Finally, an interesting aspect of this view is how, in some cases, the interaction of features (e.g. complementary or conflicting) can be seen as a variety of interaction between agents (e.g. cooperative or competitive).

4.5 Agents: Varieties of Organisation

In the discussion up to now we "told" elements whether or how to create and manipulate groups. As we increase the autonomy of elements and start talking about agents, we can leave the decision about the method of collaboration to the agents themselves. In the following we identify four different prototypical methods that agents could deploy in order to organise themselves.

Using *no groups* is the base case. Agents are structured in a pre-defined graph, and do not employ groups at all — if they need to collaborate / communicate with other agents, they do so solely by sending messages though the space (either by broadcasting, or using knowledge of the pre-configured space and routing messages to other agents). This case is not very interesting for our approach.

Going a step further, agents can *invite into their Content* others with which they want to communicate. This way, communication can be kept to a minimum while enabling agents to harness the power of the concept of identifying agents and groups. The inviting agent has control over which agents can join, as well as what messages to forward between agents in its *Content*. While it might be advantageous to have this tight control, it also means that agents have to have precisely written control structures — it is not possible to define agent wide policies (e.g. concerning which agents are allowed into

Content, or which messages to keep private and which to broadcast), as agents can easily have different tasks with different requirements.

Therefore, agents can *create external groups*, that is, an agent looking for others to help it achieve some goal can create a group, join it, and invite other agents that might be able to assist it to that group, rather its own content. That way, the inviting agent can have groups with different policies, a straightforward way to restrict messages to only reach relevant agents (namely agents within a certain group), yet it still is "in the loop", as it receives all messages that are sent between agents.

Last but not least, agents can *create internal groups*, that is, it can create a group agent within its *Content*, and invite agents to join that group[6]. The main difference between external and internal groups is that, in the latter case, the inviting agent effectively transfers control over the invited agents to the group — it does not receive communication occurring between group members, nor does it even know which agents joined the group. The inviting agent tells the group agent to solve a problem, and will be notified by the group once the problem is solved.

It should be noted here that the methods listed above present some issues that have to be solved. Firstly, whenever an agent invites others to join itself or a group, it does not know how many agents will actually react, or how long it takes the agents to join. Furthermore, agents might charge for their services, or provide solutions of different quality. While the above methods are still valid and have their respective advantages, agents will need some additional rules do decide when, and how, to choose between joining agents/services, be it through first-come-first-serve, auctions, negotiations, or other rules. Another matter that deserves close attention is group policies. In [8], several group policies specific to agents were introduced, ranging from groups where all agents can join and have no obligations to groups where only agents are allowed to join that agree to immediately honour requests from the group agent. Note that this, while similar to using different collection techniques, aims at a more fine grained level of control of the (group) agents.

4.6 Web Services

The idea behind web services is to have a collection of software services accessible via standardised protocols, whose functionality can be automatically discovered and collected together in order to provide an application working in a distributed way over the WWW. The difference between this approach and that of many (but not all) of the others above is that the services are meant to be found and grouped automatically. This search for appropriate services is supported by the idea of publishing a service's properties, for instance in XML, and then letting other agents/applets browse these published specifications in order to choose the most appropriate service [14]. Focusing on organisational matters, we can represent the initial user request as a transient group, that has to automatically find and integrate[7] a set of services to fulfil its initial goal. Once the goal is satisfied the group dies freeing all the *Content* services that it contains. Finally, in a

[6] Note that this is effectively implemented by interest groups (see Section 2.4).

[7] We disregard here all the challenges arising from a meaningful composition of services that are relevant to the Semantic Web and Ontology research communities.

number of cases, organisation of web services can be seen as corresponding to the organisation of agents, and so many of the suggestions provided for multi-agent systems above also apply to web services.

4.7 Exploiting the Group Structure

By now we have mentioned several distinct applications of the group approach, ranging from simulating object oriented behaviour over multi agent systems to feature interaction and web services. What makes our system so versatile is its ability to encode different types of meta information in the group structure. We can for example use the element structure to facilitate the communication between service robots and inhabitants of a building, by assigning a group element to each room, and grouping them in floors. Each robot joins the group of the room it is in at the moment. Furthermore, we assign agents to robots and to personal digital assistants (PDA) of the people working there. If some PDA requests coffee, the request will automatically be forwarded to the nearest robot. Also, if one robot needs help for opening the door, it will always address its nearest peer.

Other properties that can be represented in the structure are abilities of elements. The interest groups mentioned in Section 2.4, for example, create groups of elements that share the same interest / have the same abilities. We can again collect interest groups to allow agents to create hierarchical "yellow pages" that evolve within the system. Groups can also collect elements based on their ownership or affiliation.

Most obvious is the representation of tasks/subtasks through the structure. An often used example in agent technology is that of a travel agent that collects agents that deal with hotels, flights, and other aspects of a holiday. Those agents can, in turn, again make use of other, more specialised agents. Within a system where many of those travel agents are, the structure will represent the tasks that are required from them. Furthermore, defined hierarchical structures, such as in organisations and also in object oriented programming, can easily be represented.

One should note that above examples are by no means exclusive — as elements can be part of more groups, we can have several of above mentioned structures superimposed on one system. As the aims and goals of each structure may be quite disjoint from the others, it should be straightforward for elements to distinguish internally between the different types, and make full use of them.

5 Summary

In this chapter, we have provided a novel model for organising computation using group structures. These structures are both flexible and powerful. Further, since the basic framework is implemented in Java, then the computational elements can take a wide variety of forms. For example, if we consider basic Java, the elements are simply objects and so the grouping structures can provide object-oriented structures, component architectures, etc. If the elements concerned are *agents*, then grouping provides mechanisms for organising multi-agent systems, for example *teams*. And so on. By examining such instances of the model, we hope to convey the generality and flexibility of the approach. By identifying elements and groups as essentially the same entities, the model

becomes particularly simple (with correspondingly simple semantics, described elsewhere [5]). Consequently, this lightweight layer can be added to many types of system in order to provide powerful organisational capabilities.

References

1. Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, May 2001.
2. Kenneth P. Birman. The Process Group Approach to Reliable Distributed Computing. Techanical Report TR91-1216, Department of Computer Science, Cornell University, July 1991.
3. G. Booch. *Object-Oriented Analysis And Design With Applications*. Addison Wesley, 1994.
4. M. Calder, M. Kolberg, E.H. Magill, and S. Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Computer Networks*, 41(1):115–141, 2003.
5. M. Fisher and T. Kakoudakis. Flexible Agent Grouping in Executable Temporal Logic. In *Proceedings of Twelfth International Symposium on Languages for Intensional Programming (ISLIP)*. World Scientific Press, 1999.
6. S. Franklin and A. Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In J. P. Müller, M. J. Wooldridge, and N. R. Jennings, editors, *Intelligent Agents III — Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages (ATAL-96)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Heidelberg, 1996.
7. Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.
8. B. Hirsch, M. Fisher, and C. Ghidini. Organising logic-based agents. In M.G. Hinchey, J.L. Rash, W.F. Truszkowski, C. Rouff, and D. Gordon-Spears, editors, *Formal Approaches to Agent-Based Systems. Second International Workshop, FAABS 2002*, volume 2699 of *LNAI*, pages 15–27. Springer, 2003.
9. Kate Keahey. Common Component Architecture Terms and Definitions, Common Component Architecture Forum. `http://www.acl.lanl.gov/cca/terms.html`.
10. M. Plath and M. D. Ryan. Feature Integration using a Feature Construct. *Science of Computer Programming*, 41(1):53–84, 2001.
11. C. Szyperski. *Component Software - Beyond Object Oriented Programming*. Addison Wesley, 1998.
12. D. Ungar, C. Chambers, B-W. Chang, and U. Hölzle. Organizing Programs Without Classes. *Lisp and Symbolic Computation*, 4(3):37–56, 1991.
13. W3C consortium. W3C Web Services Glossary. `http://www.w3.org/TR/ws-gloss`.
14. W3C consortium. W3C Web Services Activity Statement, 2002. `http://www.w3.org/2002/ws/Activity`.
15. Peter Wegner. Classification in object oriented systems. *ACM SIGPLAN Notices*, 21(10):173–182, October 1986.
16. M. Wooldridge. *An Introduction to Multiagent Systems*. John Wiley & Sons, 2002.
17. M. Wooldridge and N. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

Adding Features to Component-Based Systems

Maritta Heisel[1] and Jeanine Souquières[2]

[1] Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik,
Institut für Verteilte Systeme, 39016 Magdeburg, Germany
heisel@cs.uni-magdeburg.de
[2] LORIA – Université Nancy2, B.P. 239 Bâtiment LORIA,
54506 Vandœuvre-les-Nancy, France
souquier@loria.fr

Abstract. Features and components are two different structuring mechanisms for software systems. Both are very useful, but lead to different structures for the same system. Usually, features are spread over more than one component. In this paper, we aim at reconciling the two structuring mechanisms. We show how component orientation can support adding new features to a base system. We present a method for adding features in a systematic way to component-based systems that have been specified according to the method proposed by Cheesman and Daniels [5].

1 Introduction

In recent years, software engineering has seen quite a number of new concepts and techniques that promise substantial contributions to a further maturing of the field. In particular, object orientation must be named here. Although object orientation did not result in as major an increase of software re-use as was expected in the beginning, it is now widely used, and it forms the basis of other very promising new approaches in software technology:

Design patterns [8] allow one to represent and re-use previously acquired problem solving knowledge. At this time, the pattern approach is widely accepted, and patterns – that need not be object oriented any more – for almost every phase of the software development process have been developed. Examples are problem frames [14], architectural styles [20], and idioms [3].

Aspect-oriented programming [16] introduces a new programming paradigm on top of object-oriented programming. The idea is to write different programs for different aspects of a software system and then use special compilers to combine the different programs into one. Aspect-oriented programming allows for better mastering the complexity of software systems.

Finally, *component-based software construction* [23, 10] has emerged from object-oriented software development. Its goal is to develop software systems not from scratch but by assembling pre-fabricated parts, as is done in other engineering disciplines. These pre-fabricated parts are called components[1]. They are independently deployable

[1] The term "component" is used differently in different contexts. Before component orientation came up, an arbitrary piece of software could be called a component. For example, in the context of software architecture, components are those units of software that perform computations (in contrast to connectors that connect components).

M.D. Ryan et al. (Eds.): Objects, Agents, and Features, LNCS 2975, pp. 137–153, 2004.

pieces of software. The most important characteristics of software components are the following:

- All services a component provides and all services it requires are accessible only through well-defined *interfaces*.
- Components adhere to *component models*. A component model is designed to allow components to interoperate that are implemented according to the standards set by the model. Building a system from components means selecting components that adhere to a particular component model and composing them in a way that is suitable to achieve the desired system behavior. Examples of component models are *JavaBeans* [21], *Enterprise Java Beans* [22], *Microsoft COM*$^+$ [18], and the *CORBA Component Model* [19].
- Components are deployed in binary form. Access to the source code of the component may not always be possible. Hence, interface descriptions play an important role in component-based development [11].

Common aspects of components and object orientation are the encapsulation of data and functionality in one unit, the role of interfaces, and the concept of instantiation. However, component models have no counterpart in object-oriented software development, which also takes it for granted that the source code of all classes is available. This is, for example, important for inheritance, which is not present among components.

In our opinion, component technology may lead to a substantial progress in our ability to develop highly complex software in high quality and in a cost-effective way.

As we have argued, it is promising to assemble software from well-specified and well-engineered components. However, this is not the only appropriate structuring mechanism for large software systems. For users of such systems, it may be more useful to structure the system according to its functionality. This structuring need not coincide with the component structure. Instead, one may look at the system as offering some basic functionality that can be augmented by *features*. According to Turner et al. [24], a feature is "a coherent and identifiable bundle of system functionality that helps characterize the system from the user perspective." Features can be identified in almost every software system, and there are even proposals to base the whole software engineering process on features [24, 6]. Moreover, language constructs for describing features have been developed [9] that support the feature engineering process.

The concepts of features and objects or components, respectively, have been developed independently of each other, and it has turned out that, usually, features are not local to one class or component of a system. Instead, the realization of features involves several classes or components. However, both features and components are adequate and powerful structuring mechanisms of software systems. Hence, it is worthwhile to try and reconcile the two approaches. Related to this goal is Zave's architectural approach to feature engineering [26, 27], called Distributed Feature Composition (DFC). DFC proposes a pipe-and-filter architecture for feature-oriented systems, where features are treated as independent components.

In this article, we demonstrate how the component structure of a system can be exploited to integrate new features into the system in a systematic way. This component structure may be arbritary and need not be an instance of some architectural style

such as pipe-and-filter. Our method relies on the work of Cheesman and Daniels [5], who use different UML notations [2] in a process that starts out from a requirements description and then identifies the necessary components and interfaces, together with their dependencies. The interface operations are specified in terms of invariants and pre- and postconditions. In performing the process, which is described in Section 2, several intermediate documents are constructed, that can be used to find out where the system has to be changed in order to integrate a new feature.

Our method to add new features to component-based systems is then presented in Section 3 and illustrated in Section 4. In Section 5, we point out what conditions must be met in order to reconcile feature orientation and component orientation.

Note that we do not consider the problem of feature interaction in this article. Feature interaction occurs when the integration of a new feature into a system leads to contradictions or unwanted or unexpected system behavior. There are numerous approaches to detect feature interactions (see [12, 4] and the literature cited there), and our feature integration method as described in Section 3 should only be applied in connection with a thorough interaction analysis[2].

2 A Method for Specifying Component-Based Systems

Cheesman and Daniels [5] propose a process to specify component-based software. This process starts from an informal requirements description and produces an architecture showing the components to be developed or re-used, their interfaces, and their dependencies. For each interface operation, a specification is developed, consisting of a precondition, a postcondition, and possibly an invariant. This approach follows the principle of *design by contract* [17]. Cheesman and Daniels' method ends with component specifications and neither considers the mapping of the developed specifications to a concrete component model nor the implementation of the specified components and interfaces.

During the application of Cheesman and Daniels' method, a number of intermediate documents (expressed in different UML notations) are generated. We will use these intermediate documents to trace the new requirements associated with a new feature in the component architecture of the system. In this respect, our feature integration method (to be described in Section 3) relies on Cheesman and Daniels' component identification and specification method. It only works when the necessary documents are present. Then, we are able to point out in which interface operations of which components changes will be necessary in order to integrate a new feature.

In the following, we summarize Cheesman and Daniels' method. In particular, we point out what documents are developed and how they depend on each other, thus allowing us to navigate among them. In Figures 1–6, we also present parts of the running example used in the book [5], namely a hotel room reservation system[3]. This example

[2] Even though the detection and elimination of feature interactions are important topics, they are not the subject of this paper. The most recent results in this area can be found in the proceedings of the *Feature Interaction Workshop*, which is held every two years.

[3] Details of this example can be found at http:www.umlcomponents.com

will be taken up again in Section 4, where we will add new features to the hotel room reservation system.

The method consists of two phases, namely *requirements definition* and *specification*.

2.1 Requirements Definition

In the requirements definition phase, a *business concept model* is set up that clarifies the notions of the application domain. It is expressed as a UML class diagram. The business concept model for the hotel reservation system is shown in Figure 1.

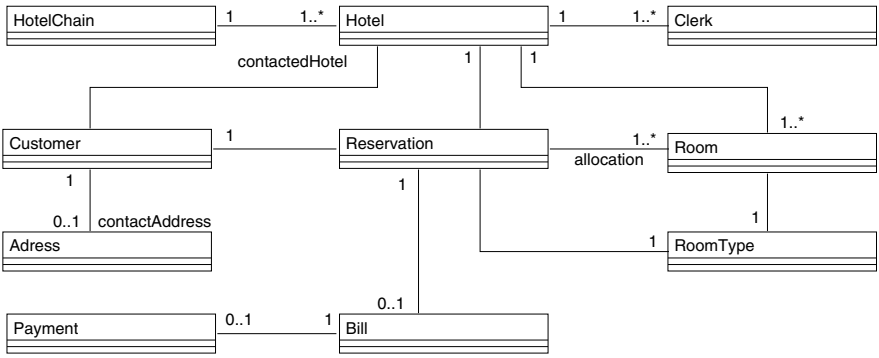


Fig. 1. Business concept model for the reservation system

Then, *business processes* relevant for the system to be constructed are expressed as activity diagrams. "Swim lanes" are used to express the responsibilities for the different steps of the processes. For each step, either an actor or the system to be constructed is responsible. On the basis of the business processes, *use cases* are identified and documented in a use case diagram.

For the hotel reservation system, we have use cases "Make a reservation", "Update a reservation", "Take up a reservation", etc., and actors "ReservationMaker", "Guest", "BillingSystem", etc. Note that the actor "BillingSystem" is not a person but an existing component that will be used by the hotel reservation system.

Each of the use cases is then described using scenarios, as shown in Figure 2. First, the main success scenario is given, which shows the case where everything works as expected. Then extensions are specified which describe alternatives or additions to the main success scenario. For example, if no room of the required type is available for the specified dates, then Step 3 of the main success scenario of Figure 2 is replaced by Step 3 of the **Extensions** section. Hence, this step is an alternative of the step given in the main success scenario. Step 3b), on the other hand, is an addition that is performed in the case that the alternative Step 3 cannot be performed successfully.

The description of the use cases concludes the requirements definition phase.

Name	Make a reservation
Initiator	Reservation Maker
Goal	Reserve room(s) at a hotel

Main success scenario

1. Reservation Maker asks to make a reservation
2. Reservation Maker selects in any order hotel, dates and room type
3. System provides price to Reservation Maker
4. Reservation Maker asks for reservation
5. Reservation Maker provides name and postcode
6. Reservation Maker provides contact email address
7. System makes reservation and allocates tag to reservation
8. System reveals tag to Reservation Maker
9. System creates and sends confirmation by email

Extensions

3. Room not available
   a) System offers alternative dates and room types
   b) Reservation Maker selects from alternatives
3b) Reservation Maker rejects alternatives
   a) Fail
4. Reservation Maker declines offer
   a) Fail
6. Customer already on file (based on name and postcode)
   a) Resume 7

Fig. 2. Scenario of the use case "Make a reservation".

2.2 Specification

This phase comprises the tasks of *component identification*, *component interaction* and *component specification*.

Component Identification. For component identification, a *business type model* is developed from the business concept model by adding further detail to the involved classes. For example, the class $Reservation$ (see Figure 1) gets the attributes $resRef : String$ and $dates : DateRange$.

In the business type model, *core types* are identified. These are the "essential" types of the application domain. They are the types that can in principle exist without association to other types. For example, a hotel can exist without a reservation, but not vice versa. For the hotel room reservation system, the core types are $Hotel$ and $Customer$.

With this information, the components to be developed can be identified. We must develop one component for the main system and one for each core type. The main system will have one interface for each use case identified in the requirements definition phase. Moreover, we must take into account those actors that are not persons but other systems. In the case of the hotel reservation system, this is the billing system. Figure 3 shows the component architecture of the hotel reservation system.

Next, the operations of each interface must be set up. These operations must allow the system to perform all the steps of the associated use case. For the interface

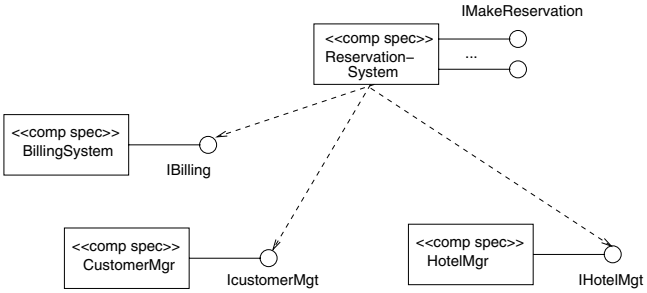


Fig. 3. Component architecture of the hotel reservation system

$IMakeReservation$, the operations are derived as follows (see Section 5.2.1 of [5]): the system must allow the person making the reservation to get details of different hotels (Step 2 of the scenario of Figure 2). Moreover, pricing and availability information must be provided for a given room type and a given date range. Finally, it must be possible to actually create a reservation (Step 7). This leads to the following operations for the interface $IMakeReservation$:

$$\begin{aligned}
&\mathit{getHotelDetails}(\mathit{in\ match} : \mathit{String}) : \mathit{HotelDetails}[\,]\\
&\mathit{getRoomInfo}(\mathit{in\ res} : \mathit{ReservationDetails}, \mathit{out\ availability} : \mathit{Boolean},\\
&\qquad\qquad \mathit{out\ price} : \mathit{currency})\\
&\mathit{makeReservation}(\mathit{in\ res} : \mathit{ReservationDetails}, \mathit{in\ cus} : \mathit{CustomerDetails},\\
&\qquad\qquad \mathit{out\ resRef} : \mathit{String}) : \mathit{Integer}
\end{aligned}$$

Component Interaction. In this phase, collaboration diagrams are developed that show how each operation of an interface of the system to be developed must interact with the other components of the component architecture in order to fulfill its purpose. These collaboration diagrams then yield the necessary interface operations of the other components (those corresponding to the core types).

Figures 4 and 5 show the collaboration diagrams for all three operations of the $IMakeReservation$ interface.

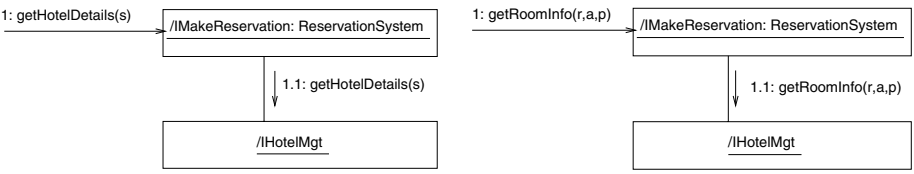


Fig. 4. Collaboration diagrams for the operations $getHotelDetails$ and $getRoomDetails$

From these collaboration diagrams, we can conclude that the interface $IHotelMgt$ must offer the operations $getHotelDetails$ (with different parameters than the op-

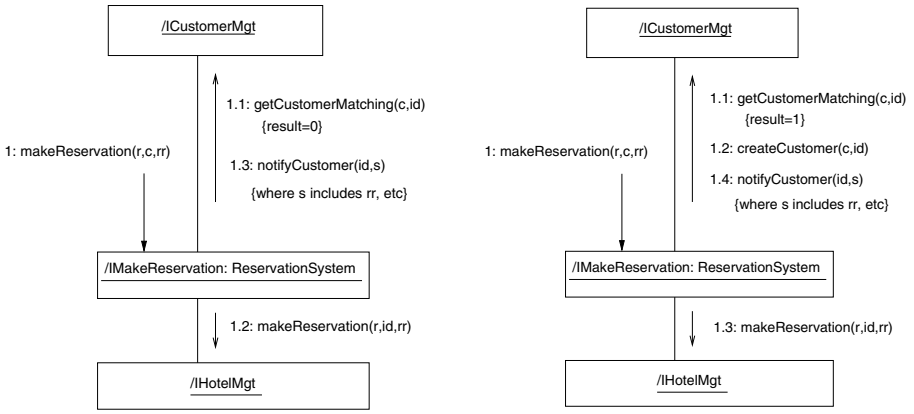


Fig. 5. Collaboration diagrams for the operation $makeReservation$

eration with the same name of the interface $IMakeReservation$), $getRoomInfo$ and $makeReser\text{-}\ vation$. The interface $ICustomerMgt$ must offer the operations $getCustomerMatching$, $notifyCustomer$, and $createCustomer$.

Component Specification. At this stage, all necessary interface operations have been identified, together with their parameters and results. In the last stage of the specification phase, the semantics of these operations is specified in terms of pre- and postconditions. The precondition expresses conditions that must be met for the operation to be successfully applied. The postcondition of an operation expresses the effect of the operation under the condition that the precondition holds. Moreover, general business rules may be expressed as invariants.

The specifications are expressed in the object constraint language OCL of UML [25]. As an example, we give the specification of the operation $MakeReservation$ of the Interface $IMakeReservation$ in Figure 6. This operation will be changed by our feature integration method in Section 4.

3 A Method to Integrate New Features into Component-Based Systems

Our method for feature integration assumes that all the documents described in the previous section are available, in particular:

- a complete class diagram
- a use case model of the main system
- scenarios describing the various use cases
- collaboration diagrams describing the interaction of the main system with other components
- specifications of all interface operations, for the main system as well as for the used components

```
makeReservation(in res: ReservationDetails,
    in cus: CustomerDetails, out resRef: String): Integer
pre:
  -- the hotel and room type specified are valid
     hotel.id -> includes(res.hotel) and
     hotel.room.roomType.name -> includes(res.roomType)
post:
  result=0 implies
       -- a reservation was created
       -- note invariant on room Type governing max no
          of reservations
       let h = hotel -> select(x | x.id=res.hotel)
          -> asSequence -> first in
         (h.reservation - h.reservation@pre) -> size=1 and
         let r = (h.reservation - h.reservation@pre)
                  -> asSequence -> first in
           r.resRef = resRef and
           r.dates = res.dateRange and
           r.roomType.name = res.roomType and
           not r.claimed and
           r.customer.name = cus.name and
           cus.postCode -> notEmpty implies
                      cus.postCode = r.customer.postCode and
           cus.email -> notEmpty implies
                      cus.email = r.customer.email
       -- result=1 implies customer not found and unable
                  to create
       -- result=2 implies more than one matching customer
```

Fig. 6. Specification of the operation $MakeReservation$ of the Interface $IMakeReservation$

Based on these documents, we can add a new feature to the main system in a systematic way. As already mentioned in the introduction, we assume that a feature interaction analysis has been performed, and that the requirements that express the properties of the new feature have been adjusted to take into account the result of the interaction analysis.

The goal of our feature integration method is to find out in which places the existing system must be modified in order to accommodate a new feature, and to give guidance how to change the documents listed above. The method starts by considering the more general documents, and then gradually proceeds to take into account the more detailed documents, such as the interface operation specifications. Figure 7 gives an overview of the method and the documents involved.

Note that our method ends with a changed set of specification documents. The feature is integrated into the existing component-based system by adjusting the implementation of the components to the changed specifications. As Cheesman and Daniels [5], we neither consider how these specifications are mapped to a concrete component model, nor how the changes in the implementation are actually performed, because these activities are context-dependent to a large extent. To adjust the implementation

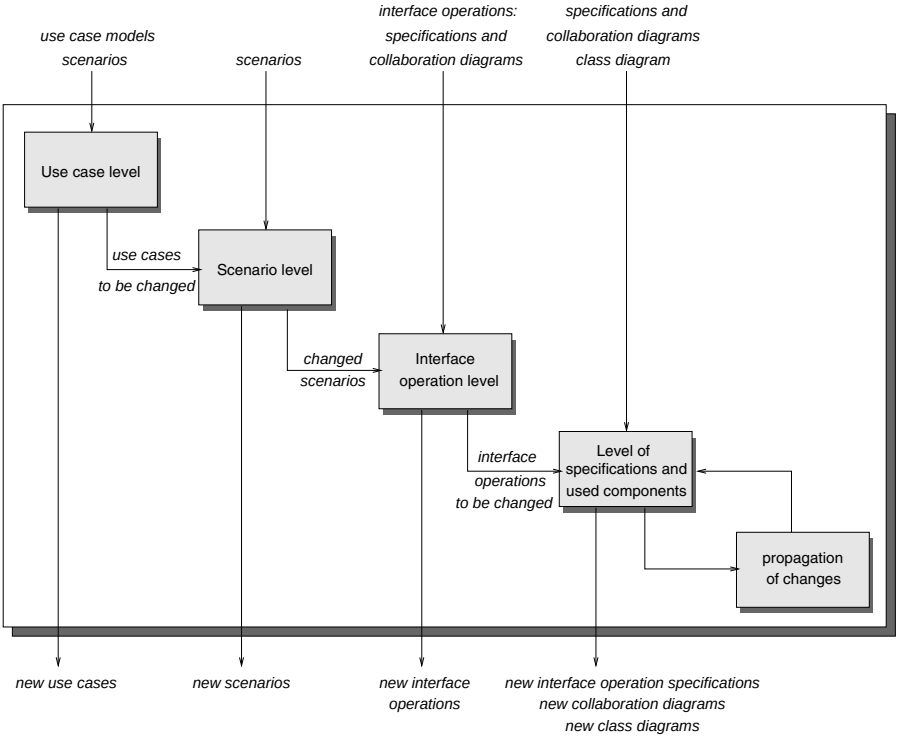


Fig. 7. Overview of the feature integration method.

to the new specification, the source code must be accessible. Hence, our method can be applied when component producer and component consumer belong to the same organization (as can be assumed to be the case for many feature-based systems, for example in telecommunications), or when the component consumer may ask the component producer to perform the required changes. In a situation where someone purchases mass-produced COTS[4] components on the market, our method is not likely to be applicable, because neither the required development documents nor the source code will be available.

Moreover, our method is not primarily intended to be used in an incremental development process such as extreme programming (XP) [1] or feature-driven development (FDD)[5] ([6], Chapter 6). We envisage more the situation where an existing system is enhanced by new features after it has been in operation for some time.

[4] Commercal off the shelf.

[5] FDD is an incremental development process, much like XP. Instead of user stories, feature sets are used to structure the iterations of the developement process. In FDD, features are defined to be "small 'useful in the eyes of the client' results" and required to be implementable in two weeks, much like user stories in XP. In telecommunications, however, features may be larger entities of functionality. Moreover, the components mentioned by Coad et al. are not components in the sense of component-based development, see the footnote in the introduction.

In the following, we describe the different steps of the method. For each step, we state which documents are needed and which documents are produced when performing the step, and we give a description of what has to be done. In Section 4, the method is applied to the hotel room reservation system introduced in Section 2.

Note that not always all output items are necessarily generated. When we say for example that the output of one step is a list of something, this list may be empty.

1. **Use case level**

   input: use case model,
   scenarios
   output: list of use cases to be changed,
   list of new use cases

   (a) First, we must decide which of the existing use cases are affected by introducing the new feature. A use case is affected if its scenarios may proceed differently than before. The use case diagram serves as a basis for this decision. If the decision cannot be made on the basis of the use case model alone, one can also have a look at the scenarios.
   (b) Second, we must decide if the introduction of the new feature leads to new use cases. This decision is based on the same documents as Step 1a. If new use cases must be introduced, then the method described in Section 2 should be applied, starting from the new use cases. Here, our method does not introduce anything new, so we will not further discuss this case in the following.

   The following steps have to be performed for each use case that must be changed.

2. **Scenario level**

   input: scenarios describing the use case to be changed
   output: new scenarios, describing the modified behavior resulting from the new feature

   In this step, we set up new scenarios by modification of the old ones. The new scenarios describe the system behavior after integration of the new feature. Modifying the scenarios may mean:

   – changing the order of actions
   – changing parameters of actions
   – introducing new actions
   – deleting actions

   For the resulting scenarios, the following steps have to be performed.

3. **Interface operation level**

   input: old scenarios,
   new scenarios as developed in Step 2,
   specifications and collaboration diagrams for the existing interface operations

output: list of interface operations to be changed,
list of new interface operations,
notes indicating how to change the global control structure of the interface implementation,
list of interface operations to delete

We must now investigate the effect of the changed scenarios on the interface operations of the use case under consideration. Four cases must be distinguished, depending on how the scenarios have been changed.

(a) If the order of actions has been changed, we must take a note that later the program that implements the global control structure of the interface must be changed accordingly. As Cheesman and Daniels' method does not require to explicitely record the order in which the different interface operations are invoked, there is no specification document which we could change to take into account the changed order of actions other than in the new scenarios. It would be suitable to introduce one more specification document that specifies in which order the different interface operations may be invoked, as is done in the Fusion method [7], for example.
(b) If parameters of actions have been changed in Step 2, then this will result in changing existing interface operations. Those interface operations that were introduced in order to take into account the scenario action in question must be changed.
(c) If new actions have been introduced into the scenarios, we must check if these can be realized using the existing interface operations (or modifications thereof), or if new interface operations must be introduced. Any new interface operation must be recorded.
(d) If actions are deleted from the scenarios, it must be checked if the corresponding interface operations are still necessary. However, operations can only be deleted if the new feature replaces the old functionality instead of supplementing it.

The next step of our method concerns those interface operations that must be changed. For the new interface operations, we proceed as described in Section 2. Again, we will not consider the new interface operations any more in the following.

4. **Level of specifications and used components**

input: list of interface operations to be changed,
together with their specifications and collaboration diagrams,
class diagram
output: new class diagram,
new collaboration diagrams and specifications of the interface operations to be changed,
list of interface operations of used components that must be changed,
list of new interface operations of used components,
list of interface operations of used components to be deleted

- For each interface operation that must be changed, we must first decide on its new parameters. To take into account parameter changes, we need the class diagram which gives information on the classes used as parameters or results of the interface operations. All changes that concern classes or attributes of classes must be recorded in the class diagram.
- Next, we consider the collaboration diagrams. If parameters of the operation have been changed, then this must be recorded in the collaboration diagram. Moreover, it must be decided if the component interactions as described by the collaboration diagram must be changed. The possible changes correspond to the changes as described in Step 2, and the collaboration diagram must be adjusted accordingly. Updating the collaboration diagrams results in lists of interface operations of used components to be changed, newly introduced, or deleted.
- Finally, the specification of each operation must be updated to take into account its new behavior.

5. **Propagation of changes**

   input: list of interface operations of used components that must be changed, together with their specifications and collaboration diagrams, list of new interface operations of used components
   output: new collaboration diagrams and specifications for the operations given as input

   Step 4 resulted in a list of new interface operations of used components and a list of interface operations of used components to be changed. The newly introduced operations are specified according to the method of Section 2, and the operations to be changed are treated as described in Step 4. This procedure is repeated until no more changes are necessary or we have reached basic components that do not use other components.

Note that after one feature has been integrated, we end up with the same – but updated – set of documents as we started out. Carrying out our method amounts to maintaining the development documents of the system. Hence, several features can be integrated successively in the same way.

4 Adding Features to a Hotel Reservation System

In Section 2, we have introduced a basic hotel room reservation system that allows its users to make reservations of hotel rooms for a given time frame. One could imagine, however, that the functionality of that system could be enhanced in several ways. These enhancements can be expressed as features.

In this section, we add the following new features to the base system.

Subscription. A customer can make several reservations at the same time, and for regular time intervals. For example, it is possible to reserve a room for two days every Monday during the next two months.

AddditionalServices. The hotel may offer additional services to be booked together with a room, for example wellness facilities.

For these features, we will carry out our feature integration method described in the previous section. We demonstrate that using our method large parts of the feature integration process can be carried out in a routine way.

4.1 Adding the Subscription Feature

We describe the steps to be carried out one by one and point out which documents are used and produced.

Step 1: Use Case Level. The starting point of our method is the use case model of the system (which we did not show in Section 2) and the scenarios describing the use cases. We have to decide which of the use cases are affected by the new feature. The use case "Make a reservation" must be changed. In the following, we will only consider that use case, although the use case "Update a reservation", for example, would also be affected. The procedure is the same for all affected use cases, however. No new use cases need to be introduced to accommodate the feature.

Step 2: Scenario Level. We now consider the scenario description given in Figure 2 and determine how it must be changed to allow for subscription reservations.

Here, a general decision must be taken. Of course, even in the presence of the subscription feature, it must still be possible to make "simple" reservations. This can be achieved in two ways: either, we define one more alternative main success scenario that describes subscription reservations. This is possible, because one use case can have more than one associated success scenario. Or, we define the modified behavior as extensions of the main success scenario of Figure 2. For the purposes of this paper, we decide to take the second option.

Hence, the following modified or new actions are introduced in the **Extensions**-section of the scenario description. Steps 1 and 3 are modified, and a new Step 2a is added:

1. Reservation Maker asks to make a *subscription reservation*
2a. Reservation Maker provides subscription details

The text of Step 3 remains the same, but we must make a note that the procedure to calculate the price of the reservation must probably be modified.

Step 3: Interface Operation Level. Based on the modified scenario constructed in the previous step, we must decide how the introduced changes affect the operations of the interface $IMakeReservation$. We cannot delete any operation, but we need a new one called $enterSubscriptionDetails$. This function, which we must specify according to the method of Section 2[6], will yield an object containing all details necessary for a subscription reservation.

[6] As the specification of interface operations is not part of our method, we will not present specifications of newly introduced operations in this paper. We only demonstrate how changes are made in a systematic fashion.

Of the existing operations, we decide that $getHotelDetails$ and $getRoomInfo$ may remain unchanged, whereas $makeReservation$ must be changed.

Step 4: Level of Specifications and Used Components. We must now update the documents describing the operation $makeReservation$, which has the following profile:

makeReservation(*in res* : *ReservationDetails*, *in cus* : *CustomerDetails*,
out resRef : *String*) : *Integer*

We see that the change manifests itself in the parameter type $ReservationDetails$, which must be changed. In the class diagram (see Figure 1), the class $Reservation$ must be adjusted. This class had got an attribute $dates : DateRange$. This attribute must now contain a nonempty collection of date ranges instead of one date range. We change the attribute declaration accordingly: $dates[1..*] : DateRange$.

Next, the collaboration diagrams of the operation $makeReservation$ (see Figure 5) must be considered. The collaboration diagram itself need not be changed. However, the parameter r of type $ReservationDetails$ is passed on to the operation $makeReservation$ of the interface $IHotelMgt$. Hence, this operation must be checked for necessary changes, too.

Finally, we must update the specification of $makeReservation$ given in Figure 6. An inspection shows that all the changes are hidden in the res of type $Reservation Details$. This concerns the line `r.dates = res.dateRange` of the specification. Hence, the text of the specification remains the same.

Step 5: Propagation of Changes. The changes performed in the previous steps must be propagated in the documents concerning the operation $makeReservation$ of the interface $IHotelMgt$, as was found out in Step 4. We do not present this propagation in the present paper.

This concludes the integration of the subscription feature into the hotel room reservation system. We did not introduce any new use cases, but we introduced new actions in the scenario of an affected use case, which resulted in a new interface operation. Another interface operation was changed, but the change concerned only one of the types involved in the operation.

4.2 Adding the AdditionalService feature

We now integrate this second feature, starting from the same set of documents as in the Section 4.1. Note that for reasons of simplicity, we add the feature to the base system and not to the system that results in having added the subscription feature.

Step 1: Use Case Level. As in Section 4.1, we judge that no new use cases are necessary, and that the use case "Make a reservation" is affected by the billing feature.

Step 2: Scenario Level. We change the scenario of Figure 2 by introducing a new step in the **Extension** section:

2a. Reservation Maker selects additional services

As in Section 4.1, the text of Step 3 remains the same, but we must make a note that the procedure to calculate the price of the reservation must probably be modified.

Step 3: Interface Operation Level. As far as the operations of the interface $IMake\text{-}Reservation$ are concerned, we can certainly not delete any of them. We decide that information on the additional services provided belongs to the contacted hotel. Hence, we need no new interface operation, but we must change the operations $getHotelDetails$ and $makeReservation$. The operation $getRoomInfo$ remains unchanged.

Step 4: Level of Specifications and Used Components. To take into account that additional services are associated with a hotel and can possibly be associated with a reservation, we must update the class diagram by introducing a class $AdditionalServices$ with associations to the classes $Hotel$ and $Reservation$, as shown in Figure 8.

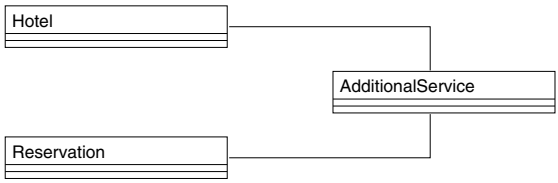


Fig. 8. Part of new class diagram for the hotel reservation system

We now consider the updates necessary for the operation $getHotelDetails$. The parameter of $getHotelDetails$ does not change, and also the collaboration diagram remains the same as the one shown in Figure 4. However, we must note that the operation $getHotelDetails$ of the interface $IHotelMgt$ must be changed. This operation must now also make available information about the additional services provided by the hotel. Since we have not presented the specification for $getHotelDetails$ in Section 2, we do not discuss the changes of the specification here.

The changes that are necessary for the operation $makeReservation$ resemble the ones described in Section 4.1. The reservation details must be changed to take into account a possible reservation of additional services.

Step 5: Propagation of Changes. The changes performed in the previous steps must be propagated in the documents concerning the operations $getHotelDetails$ and $make\text{-}Reservation$ of the interface $IHotelMgt$, as was found out in 4. We do not present this propagation in the present paper.

When integrating the additional services feature, we neither introduced new use cases nor new interface operations. However, we changed the class diagram substantially, and consequently had to change two of the three existing interface operations. The most substantial changes, however, are the ones to be performed in the $HotelMgr$ component.

This shows that it can hardly be avoided that several components are affected by introducing a new feature. However, by tracing the development documents as we have demonstrated in this section, we do not get lost in the component structure, but are guided to the components that must be changed.

5 Conclusions

As discussed in Section 1, both features and components are powerful and adequate structuring mechanisms for software systems. However, they lead to different system structures. This is due to the fact that components structure the system from a developer's point of view, whereas features structure the system from a user's point of view. Of course, both of these views are important and should be used when describing software systems.

This dichotomy leads us to the question how both structuring mechanisms can be used in parallel without incoherences. The present paper proposes an approach how to reconcile the two structuring mechanisms of features and components. On the one hand, the systems we consider are structured according to a component architecture. This structure reflects the implementation units of the system. On the other hand, we consider our systems to consist of a base system to which features can be added. This second structure, however, is not directly reflected in the implementation of the software system.

To bridge the gap between the two system structures, we need to map the feature structure onto the component structure. In particular, this means that we must *trace* each feature in the component structure. This approach is similar to requirements tracing as performed in requirements engineering [15]. To allow for the tracing, a number of intermediate documents are necessary, together with the dependencies between them (see also [13]). For example, scenarios are derived from use cases, interface operations are derived from scenarios, and collaboration diagrams are derived from interface operations. Thus, following the dependency links between the different documents allows us to determine how the realization of a feature is distributed in the component architecture. Integrating a feature is then performed by following those paths in the dependency structure where changes occur. As we have shown in Sections 3 and 4, this traversal of the dependency structure is possible in a systematic way.

Hence, our method does not change the fact that features are distributed over several classes or components, but it helps to deal with this fact in a satisfactory way.

Acknowledgments

We thank Hung Ledang and Thomas Santen for their comments on this paper.

References

1. Kent Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, 1999.
2. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.
3. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns.* John Wiley & Sons, 1996.
4. Muffy Calder and Alice Miller. Detecting feature interactions: how many components do we need? In H.-D. Ehrich, J.-J. CH. Meyer, and M.D. Ryan, editors, *Objects, Agents and Features. Structuring mechanisms for contemporary software*, this volume. Springer-Verlag, 2004.

5. John Cheesman and John Daniels. *UML Components – A Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2001.
6. Peter Coad, Eric Lefebvre, and Jeff De Luca. *Java Modeling In Color With UML.* Prentice Hall, 1999.
7. D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The Fusion Method.* Prentice-Hall, 1994.
8. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison Wesley, Reading, 1995.
9. Stephen Gilmore and Mark Ryan, editors. *Language Constructs for Describing Features.* Springer-Verlag, 2000.
10. George T. Heineman and William T. Councill. *Component-Based Software Engineering.* Addison-Wesley, 2001.
11. Maritta Heisel, Thomas Santen, and Jeanine Souquièeres. *Toward a formal model of software components. In Chris George and Miao Huaikou, editors, Proc. 4th International Conference on Formal Engineering Methods*, LNCS 2495, pages 57–68. Springer-Verlag, 2002.
12. Maritta Heisel and Jeanine Souquièeres. A heuristic algorithm to detect feature interactions in requirements. In Stephen Gilmore and Mark Ryan, editors, *Language Constructs for Describing Features*, pages 143–162. Springer-Verlag, 2000.
13. Maritta Heisel and Carsten von Schwichow. A method for guiding software evolution. In *Proceedings IASTED Conference on Software Engineering (SE 2004)*, 2004. to appear.
14. Michael Jackson. *Problem Frames. Analyzing and structuring software development problems*. Addison-Wesley, 2001.
15. Matthias Jarke. *Requirements tracing. Communications of the ACM*, pages 32–36, December 1998.
16. Gregor Kiczales. *Aspect oriented programming. ACM SIGPLAN Notices*, 32(10):162–162, October 1997.
17. Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, second edition, 1997.
18. Microsoft Corporation. COM^+, 2002. http://www.microsoft.com/com/tech/COMPlus.asp.
19. The Object Mangagement Group (OMG). *Corba Component Model, v3.0*, 2002. http://omg.org/technology/documents/formal/components.htm.
20. Mary Shaw and David Garlan. *Software Architecture. Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.
21. Sun Microsystems. *JavaBeans Specification, Version 1.01*, 1997. http://java.sun.com/products/javabeans/docs/spec.html.
22. Sun Microsystems. *Enterprise JavaBeans Specification, Version 2.0*, 2001. http://java.sun.com/products/ejb/docs.html.
23. Clemens Szyperski. *Component Software*. ACM Press, Addison-Wesley, 1999.
24. C. Reid Turner, Alfonso Fuggetta, Luigi Lavazza, and Alexander Wolf. A conceptual basis for feature engineering. *Journal of Systems and Software*, 49(1):3–15, 1999.
25. Jos Warmer and Anneke G. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1999.
26. Pamela Zave. Architectural solutions to feature-interaction problems in telecommunications. In K. Kimbler and W. Bouma, editors, *Proc. 5th Feature Interaction Workshop*, pages 10–22. IOS Press Amsterdam, 1998.
27. Pamela Zave. Feature-oriented description, formal methods, and DFC. In Stephen Gilmore and Mark Ryan, editors, *Language Constructs for Describing Features*, pages 11–26. Springer-Verlag, 2000.

Components, Features, and Agents in the ABC

Tiziana Margaria[1,2]

[1] Universität Göttingen, Lotzestr. 16–18, 37083 Göttingen, Germany
margaria@cs.uni-goettingen.de
[2] METAFrame Technologies GmbH, Borussiastr. 112, 44149 Dortmund, Germany
TMargaria@METAFrame.de

Abstract. In this paper, we show how the concepts of objects, components, features and agents are used today in the *Application Building Center* (ABC) environment in order to marry the modelling of functionally complex communication systems at the application level with an object-oriented, component based implementation. Characteristic of the ABC is the coarse-grained approach to modelling and design, which guarantees the scalability to capture large complex systems. The interplay of the different features and components is realized via a coordination-based approach, which is an easily understandable modelling paradigm of system-wide business processes, and thus adequate for the needs of industrial application developers.

1 Motivation

In a scenario of increasingly complex, distributed, and interdependent service provision, moving large portions of the needed application programming load from programming experts to application experts (or even to end-users) becomes a major challenge. For application experts, this requires freeing application design and largely also development activities from their current need of programming expertise. These observations were the starting point for our development of METAFrame Technologies' *Application Building Center* ABC [21], a tool for graphical, library-based application development. As explained in [13], the choice of modelling applications at coarse granularity in combination with a 'simple' component and coordination model, which support easy component reuse, were central for the success of the approach, measured in terms of a seamless integration of our technology into the customers' processes. This success was particularly due to the choice of managing the inter-component or inter-tool flow of control in a programming-free way.

The concrete application scenario considered in this paper, the development of complex collaborative internet services, combines heterogeneous architectures with black/grey-box implementation, one of the typical difficulties of legacy systems, in particular under continuous redesign and modification during the lifetime of the systems [25]. The ABC is used there for the definition and enforcement of complex behaviors (features) which are *superposed* on and coordinated within the system under development. The challenge is precisely how to handle

M.D. Ryan et al. (Eds.): Objects, Agents, and Features, LNCS 2975, pp. 154–174, 2004.

this superposition and coordination in an understandable, well partitioned, and manageable way:

- it should be *expressive* enough to capture coordination tasks like steering external components and test tools, reacting to events, measuring responses, and taking decisions that direct the control flow within a distributed system,
- it should be *non-intrusive*, i.e. we cannot afford having to modify the code of subsystems, and this both for economical reasons and lack of feasibility: most components and external applications are in fact black boxes,
- it should be *intuitive and controllable* without requiring programming skills. This implies that we need a lightweight, possibly graphical approach to the design of the coordination of activities (be they services, features, or single functionalities) and that easy and timely validation of coordination models should be available.

In the ABC, we use coordination graphs [13] (explained in Sect. 2) as a basis for the definition of *agents* that make services (expressed as interoperating collections of *features*) available at runtime to the single users. Coordination graphs provide an adequate abstraction mechanism to be used in conceptual modelling because they direct developers to the identification and promotion of user-level interactions as first-class entities of application development, a pre-condition for taming the complexity of system construction and evolution. In our solution, we have adopted a coarse-grained approach to modelling system behavior, that accounts for the required simplicity and allows direct animation of the models as well as validation (via model checking) directly at the application level.

Our Application Profile

Our approach addresses scenarios where the *user-level flow of control* is of central importance and it is realized via coordination graphs. They establish a specific modelling level which allows one to directly model these control aspects on their own, without being overwhelmed by implementation aspects like, e.g., data structures, architectures, deadlocks, and load balancing. These aspects are hidden to the application designer and taken care of at a different level, during the object-oriented, component-based development of the single functionalities (which may well have a global impact on the system), capturing user-level requirements. Particularly well-suited for our approach are therefore applications where the flow of control frequently needs adaptation or updating, as it is the case when addressing user-specific workflows or situation-specific processes. Besides the test of CTI applications presented in [5], we also successfully addressed the design and test of advanced, role-based, client-server applications like the Online Conference Service [11, 12] that we are using as a running illustrative example in this paper.

Applications with a very stable or trivial user level flow of control do not profit from our coordination-based approach. In particular, this concerns highly algorithmic applications with intensive internal computation. Such applications

may well constitute individual building blocks in our setting, but they are inadequate for a coordination graph-based development.

In this paper, we first precise our understanding of features (in Sect. 2, then we introduce in Sect. 3 our concrete example: the Online Conference Service (OCS), and in Sect. 4 we present the organization of ABC applications and their relation to the concepts of components, objects, feature and agents. Subsequently, we describe in detail the adopted component model and its use (Sect. 5), sketch the resulting coordination-driven execution in Sect. 5.4), and illustrate the interplay of the described concepts in Sect. 6 on a specific portion of the OCS. Finally Sect. 7 contains our conclusions.

2 Our Understanding of Features

There are many definitions of features, depending heavily on their context and their use. Our understanding of the feature concept can be well explained along the similarities and differences wrt. the definitions of feature and of feature-oriented description given earlier in this volume in [3]. Although we too learned to know and appreciate the concept and the use of features in the context of Intelligent Networks [7, 8, 23], our notion of features is more general in order to also capture a more general class of services like online, financial, monitoring, reporting, and intelligence services:

Definition 1 (Feature).

1. *A* feature *is a piece of (optional) functionality built on top of a base system.*
2. *It is* monotonic*, in the sense that each feature* extends *the base system by an* increment *of functionality.*
3. *The description of each feature* may *consider or require other features, additionally to the base system.*
4. *It is defined from an* external *point of view, i.e., by the viewpoint of* users *and/or* providers *of* services*.*
5. *Its granularity is determined by* marketing *or* provisioning *purposes.*

Differently from the IN setting, where the base system was the switch, offering POTS functionality, and the features were comparatively small extensions of that behaviour, we have (e.g. in CSCW-oriented internet services like the OCS) a *lean* basis service, that deals with session, user, and role-rights management, and a rich collection of features.

Other than in [3], where features are understood as modifiers of the basic service, in order to account for complex evolutions of services, we allow a *multi-level organization* of features, whereby more specialistic features build upon the availability of other, more basic, functionalities.

In order to keep this structure manageable and the behaviours easily understandable, we restrict us to *monotonic* features, which are guaranteed to add

behaviour. Restricting behaviour, which is also done via features in other contexts (e.g. [6]), is done in an orthogonal way in our setting, via constraints at the requirements level.

Additionally, we distinguish between features as implementations and properties of feature behaviours. Both together give the feature-oriented description of services enforced in the ABC.

Definition 2 (Feature-Oriented Description).

1. *A* feature-oriented service description *of a complex service specifies the behaviours of a* base system *and a set of* optional features.
2. *The behaviour of each feature and of the basic system are given by means of Service Logic Graphs (SLGs) [8].*
3. *The realization of each SLG bases on a library of* reusable components *called Service Independent Building-Blocks (SIBs).*
4. *The feature-oriented service description includes also a set of* abstract requirements *that ensure that the intended purposes are* met.
5. Interactions *between features are regulated* explicitely *and are usually expressed via* constraints.
6. *Any* feature composition *is allowed that does not violate any constraint.*

In contrast to the proposal by [3], we distinguish the description of the feature's behaviour from that of the legal use of a feature. Restrictions to behaviours are in fact expressed at a different level, i.e. at the requirements level, and they are part of an aspect-oriented description of properties that we want to be able to check automatically, using formal verification methods.

3 Application: The Online Conference Service (OCS)

The OCS (Online Conference Service) (see [11] for a description of the service and of its method of development) proactively helps authors, Program Committee chairs, Program Committee members, and reviewers to cooperate efficiently during their collaborative handling of the composition of a conference program. It is customizable and flexibly reconfigurable online at any time for each role, for each conference, and for each user. The OCS has been successfully used for over 25 computer science conferences, and many of the ETAPS Conferences. This year it served them all, with 6 instances of the service running in parallel.

The service's capabilities are grouped in *features*, which are assigned to specific *roles*. In the OCS, a single user may cover many roles (e.g., PC Members may submit papers and thus be simultaneously Authors), and can switch between them at any time during a working session. A fine granular roles and rights management system takes care of the adequate administration of the context, role and user-specific permissions and restrictions. The different roles cooperate during the lifetime of a PC's operations and use the OCS capabilities, which are provisioned at the feature level. Through the cooperation of its features, the OCS provides a timely, transparent, and secure handling of the papers and of the related submission, review, report and decision management tasks.

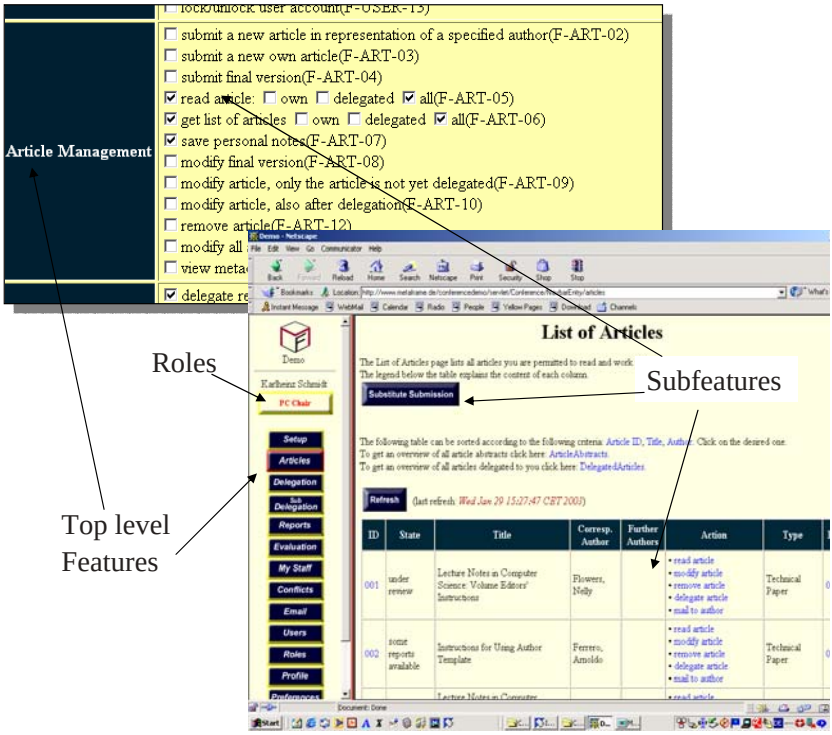


Fig. 1. Role-based Feature Management in the OCS

3.1 Feature Description

Features are assigned to the roles, and can be fine-granularly tuned for conference-specific policies. E.g., some conferences practice blind reviewing, meaning that certain fields of the article submission form are not published to the reviewers and secret between Author and PC Chair. In this paper we focus on the principal features and on the discussion of their implications in the ABC-based service development. The following three features are illustrative of the size and granularity adopted in the OCS, while the full collection is shown in Fig. 5.

1. **Article Management:** Over 30% of the service activity consistently concerns this feature. The central page corresponding to this feature is the Article overview (Fig. 1(bottom)), which also contains links to activities like report submission or paper delegation that go beyond just providing access to the article and article managements' pages.
2. **Delegation Management:** this feature allows the PC Chair to delegate papers to appropriate PC Members and supports PC members in their dialogue with their subreviewers. It manages the PC Member's and Reviewer's tasklists. The supported delegation process is iterative as PC members/subreviewers might refuse a task, e.g., due to conflicts of interest and/or lack of expertise.

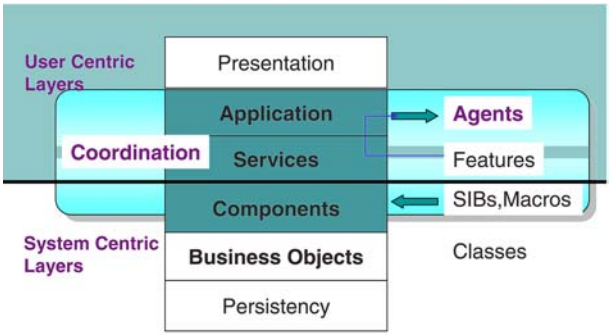


Fig. 2. The Layered Architecture of the ABC Applications

3. **Setup Management:** this feature enables the responsible administrator and/or the PC Chair to configure the service before it goes public. It also allows online reconfigurations (e.g. setting global deadlines, assigning further responsibilities, establishing newsgroups) while the service is running.

As shown in Fig. 1, the features interact: by configuring differently the role PC Member (Feature Role Management, Fig. 1(top)) a PC Chair can at any moment grant and revoke coarse and fine granular access rights to the whole Article Management feature or to portions of it to single users or to user groups. We address the challenge to guarantee that these dynamically defined possible service behaviours obeye all our requirements.

3.2 Property Description

Several security and confidentiality precautions have been taken, to ensure proper handling of privacy and intellectual property sensitive information. In particular,

- the service can be accessed only by registered users,
- users can freely register only for the role Author,
- the roles Reviewer, PC Member, PC Chair are sensitive, and conferred to users by the administrator only,
- users in sensitive roles are granted well-defined access rights to paper information,
- users in sensitive roles agree to treat all data they access within the service as confidential.

We need to be able to check these service-wide properties in a service architecture organized in a hierarchical feature structure. The following sections explain how the ABC has been designed and organized along these needs of the development of services like the OCS.

4 The Structure of the ABC Coordination Framework

As shown in Fig. 2, we have refined the typical modern layered application architectures: in order to support division of labour and to better integrate

application experts, customers, and even end user in the development process, we added three layers bridging the gap between the persistency and business objects layers on the one side and the presentation layer on the other side. Besides opening part of the application development to people without programming expertise, this refined architecture may also clarify terminology. Let us traverse the layered architecture of ABC applications bottom up, starting with the

Business Objects Layer:
This layer establishes and defines the "things" that are dealt with in the application, like article, user, delegation, report in the OCS application. Since we develop internet-based applications within an object-oriented paradigm, at this layer we define the classes and the objects that live and act in the application.

Business objects are an application-specific matter. They can be exchanged, reused and extended within a homogeneous class of applications. E.g., the OJS, the companion service of the OCS for the management of the refereeing process for journals, shares most of the OCS business objects "as they are", with some specialization only needed for selected things that take different or new responsibilities. However, other, more technical business objects, like Role Manager or Security Handler can well be used far beyond the scope of a given application.

Splitting the 'classical' Application layer on top of the Business Objects layer, leads to the following three (sub-)layers, which are characteristic for the ABC: the system-centric Components Layer and the user-centric Services Layer and Applications Layer. These are the layers we address in the reminder of the paper.

Components Layer:
In the ABC, components capsulate functionality, realized by means of the business objects, in a way that allows a coordination centric development as described in in Sect. 5.4. Following [2], this approach to development helps developing complex (information) systems in a structured way, with two main properties that are also valid of the ABC:

- **externalization** of the business rules, i.e., of the interoperation with and dependencies of the business processes from the underlying computations (which are capsulated in reusable components)
- **encapsulation** of parts of the user-oriented behaviour into first class entities of the application design, which exist and evolve independently of the underlying computations, and follow the evolution of the business context they implement rather than the technical development of the software system.

Components in the ABC are called *SIBs* (Service Independent Building Blocks). They may be hierarchically structured using a *macro* facility. Macros allow developers to group subservices into higher-order components that can be consistently reused as if they were (basic) SIBs (Sect. 5). In fact, the simple interfacing structure makes them easily reusable even beyond a specific application domain.

Whereas the development of basic SIBs is a programming activity, developing macros is already a coordination activity, technically supported in the ABC by the Service Logic Graph editor, as shown in Figs. 3 to 6.

Services Layer:
The Services layer defines the palette of coarse-grained services offered to the users, and implements them in terms of the available components. In the ABC at the service level we explicitly speak of *Features*, like the Setup, Delegation, Articles management features accessible in the OCS to (some of the) end-users. Features are explicitly made available to the outside over the GUI, as we can see in the navigation bar on the left side of the screen shots of Fig. 1.

Like SIBs, also Features can be hierarchically structured in terms of Service Logic Graphs: in contrast to SIBs, however, where the hierarchical structuring is simply a bottom-up developement support, the substructuring of Features is a central means for organising the complex and multidimensional, personalized role-right-context management.

Again, like SIBs, features are also shareable and reusable across applications and, in principle, although it makes more seldom sense, even across application domains (Sect. 6).

Applications Layer:
The Application layer defines the coarse structure of the workflows of the various potential users in their specific roles in terms of a global SLG coordinating the involved features. In particular, it defines user/situation-specific Agents via

- workflows that are enabled only at *certain times* (e.g. the submission of reports makes sense only between certain deadlines),
- or for *certain roles* (e.g. the configuration of the whole OCS service for a conference, which is in the hands of the PC Chairs),
- or under *certain circumstances* (e.g. the update of a tasklist for a PC Member follows a delegation act of a paper to this member, the discussion in an online forum is enabled only during the discussion phase, only if the PC Chair has opened this forum, and only if the PC Member is not banned via conflicts from accessing this particular forum).

In fact, at runtime, a personal *Agent* is associated to each user and for each session: the agent is in charge of navigating the Application level service Logic Graph for this user in the current context, i.e., taking appropriate care of the current user profile, and the time- and configuration-variable limitations due to roles, deadlines, conflicts, enabled or disabled features, history etc...

We consider the Services and Applications Layers both as *User-centric*, since the responsibility for their shaping, development, and maintenance is not of a software-technological kind, but rather of a domain-specific, application driven character, and therefore they should be in the hands of application experts, customers and even end users, who are responsible for defining the requirements to the application and also provide feedback at this level during its entire lifecycle. As explained later in the following sections, all the user-centric portion of application development is based on a business-process oriented point of view and formalized in a way similar to flow-graphs. Great care has been taken to ensure that in the ABC no knowledge of programming, in particular no knowledge of object-orientation is necessary at these higher levels.

Software engineers and/or programmers should be familiar with these layers, and be involved here mainly from the point of view of the consistency and structuring between the service-level view of the system and the lower, system-centric view. In fact, since features are themselves given via a collection of collaborating components that are implemented within an object oriented paradigm, the consistency between the two views is a central requirement for the whole design.

5 Coordination-Oriented Component-Based Development

The coordination-oriented approach to component-based application development adopted in the ABC has been described in technical detail in [13] and we re-examine it here from the point of view of its usefulness for the feature-based development of services and applications.

As already introduced, elementary components in the ABC are called SIBs (Service Independent Building blocks), and complex behaviours are defined in terms of so called Service Logic Graphs (SLGs). The choice of these names is an intentional analogy to the terminology of the Intelligent Network Services application domain [7], and stems from our previous experience in building a service definition environment for IN services [23]. Particularly successful was there the definition of a *global functional plane* for IN services, which defined the *business logic* of distributed services in terms of a collection of standardized, reusable building blocks (the SIBs) and a collection of superordinate features.

In the ABC we capsule in components well-defined pieces of functionality that

- are exchangeable within the service,
- can be reused for other services of the same sort (i.e., internet applications, computer-telephony applications) or across the sorts (e.g. CVSCreateConnection, or check_pwd, which are general purpose components),
- and can be deployed independently within the environment they are intended for (e.g. Apache or Tomcat, Java Beans Environment, or proprietary telephony switch environments).

In other words, as advocated in a component based development and consistent with the adopted coordination-oriented application development, they have a controlled autonomous life within the target environment. Additionally, like IN SIBs, they provide "meaningful" functionalities in the application domain. The identification and management of components is thus one of the central concerns in the ABC development process, and it is a joint concern of the programmers and application experts.

Concretely, the ABC supports at the component level

- a basic granularity of components in term of *SIBs* which offer atomic functionalities and are organized in application-specific collections. These building blocks are identified on a functional basis, understandable to application experts, and usually encompass a number of 'classical' programming units (be they procedures, classes, modules, or functions).

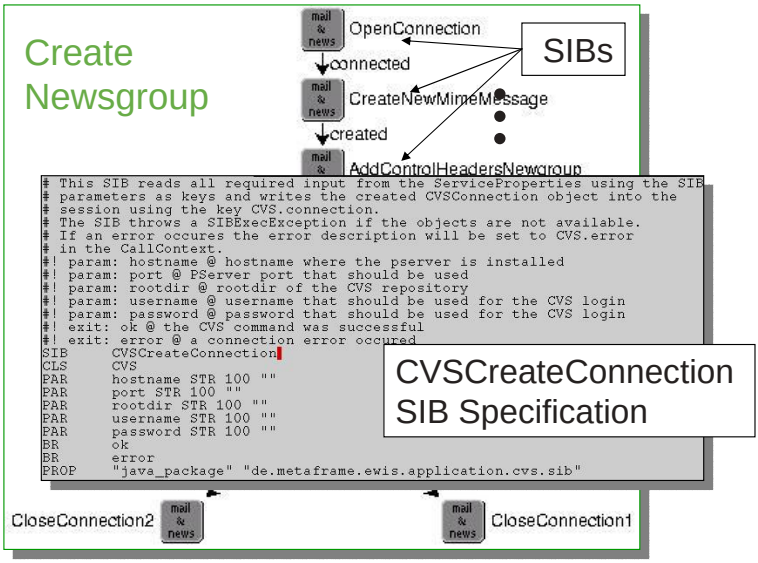


Fig. 3. SIB Occurrences in a SLG, and SIB Specification

– and a structuring mechanism via *macros*, which allow developers to build higher-order components that can be consistently reused as if they were basic components. Consistency is in fact a central concern in terms of analyzability and diagnosis via model checking - as explained briefly in Sect. 5.5.

Application development consists then of the behaviour-oriented combination of SIBs and macros on a *coarse*-granular level.

5.1 The SIB Component Model

The ABC has a very simple **component model**: as shown in Fig. 3 for the CVSCreateConnection SIB, which implements the functionality of connecting to a CVS repository[1], it has

1. a *name*, here CVSCreateConnection, characterizing the SIB,
2. a *category* characterizing the tool, in this case the CVS versioning system, subsystem, or – for test case-specific blocks – the specific management purpose (e.g. report generation) it relates to,
3. a set of *formal parameters* that enable a more general usage of the block. Here the parameters are the hostname, port, rootdir, username and password needed to establish a connection to CVS,
4. a set of *branches* which steer the execution depending on the results of the execution of this SIB. Here, a connection attempt can either terminate successfully and take the OK branch or fail and take the error branch, and
5. *execution code* written in the coordination language, typically to wrap the actual code that realizes the functionality.

[1] CVS is a well established concurrent versioning system [4].

The *name, category, formal parameters*, and *branches* of this component model provide a very abstract characterization of the components. This will be used later to check the consistency of coordination graphs. The computational portion is encapsulated in the execution code, which is independent of the coordination level, thus it is written (or, as in this application, generated) once and then reused across the different scenarios.

5.2 Coordination Models

In contrast to other component-based approaches, e.g., for object-oriented program development, the ABC focusses on the dynamic behavior: (complex) functionalities are graphically stuck together to yield flow graph-like structures (the SLGs) embodying the application behavior in terms of control, as shown in Fig. 4 for a portion of the OCS. These SLGs constitute therefore directly our coordination models, and they are at the same time coarse-grained formal models of the business logic of the applications. As explained in [13], they are internally modelled as labelled Kripke Transition Systems [16] whose nodes represent (elementary) SIBs and whose edges represent branching conditions.

Through this non-standard abstraction in our model we obtain a **separation of concerns** between the control-oriented coordination layer, where the application designer is not troubled with implementation details while designing or evaluating the applications, and the underlying data-oriented communication mechanisms enforced between the participating subsystems, which are hidden in the SIB implementation. Our tools support the automatic generation of SIBs according to several communication mechanisms (CORBA [17], RMI [24], SOAP [19], and other more application-specific ones), as described e.g. in relation to the ETI (Electronic Tool Integration) platform application [22, 14].

5.3 Hierarchy and Macros

The ABC supports a truly hierarchical design at the coordination level. Within an abstraction step, a (part of a) coordination model (an SLG) can be stored as a *macro*, which this way directly becomes available as a new generic SIB of a particular category macro. Beside the identifier and the name of the macro, the formal parameters and the outgoing branches have to be specified. The parameters of the macro can be mapped to (selected) parameters of the underlying SIBs. Similarly, the set of (un-set) outgoing branches of the underlying SIBs defines the outgoing branches of the macro (see [23] for a detailed discussion). Fig. 4 shows how this works in practice: the SLG here reported is itself reusable macro, and in fact they appear as such in the SLG of Fig.6.

As usual, the resulting hierarchy is a design and management aid without any influence on the execution time: during the execution, macros are automatically unfolded (concretized) and the underlying graph structure is executed – this also explains the choice of the term macro for our hierarchy mechanism.

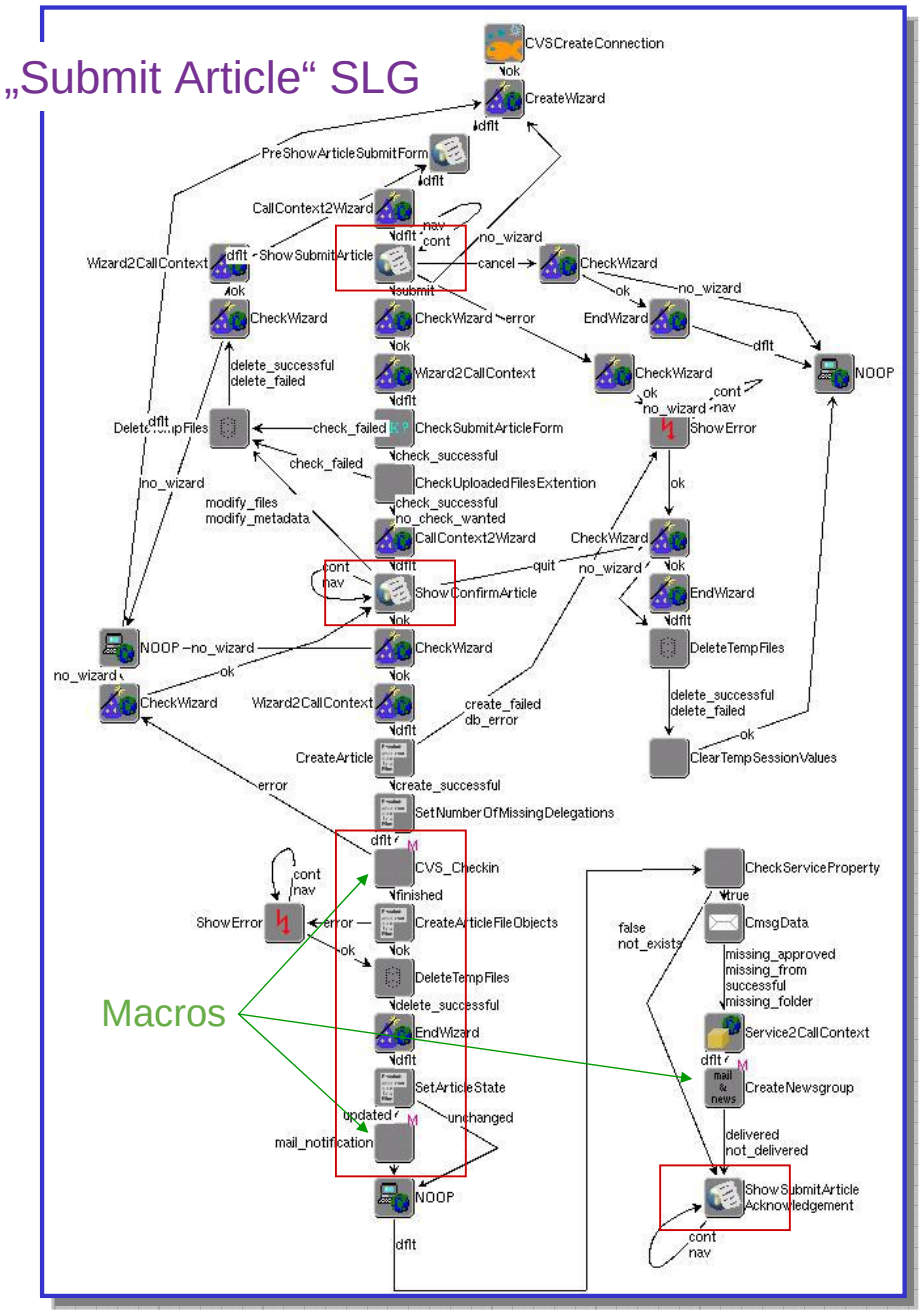


Fig. 4. A Hierarchical Macro: The Submit Article Feature

5.4 Coordination-Driven Execution

From the execution point of view, the actually executed coordination sequence is a path in the SLG and it is determined at runtime by the outcomes of the execution of the actual SIBs.

In the concrete case of the Submit Article macro of Fig. 4, several SIBs correspond to check-points that steer the further execution flow according to some decisions: for example, SIBs like *CVSCheckin* or *CheckSubmArticleForm* check the correctness of central elements or parameters in an execution. They answer questions like "is the File upload in the versioning system correctly executed" or "given a submitted article form, are its required fields correctly filled"?

The possible answers are coarsely grouped in terms of outgoing branches of the respective SIBs. As an example, the SIB *CheckSubmArticleForm* can respond that the required data for a submission are OK (branch *check_successful*) or not (branch *check_failed*). Thus the branching lifts the results of relevant (parameter) checks, and only them, to the coordination level, makes them visible to the designer, and makes them also accessible to the model checker as explained in the next Section. Depending on the outcome of the checks, one of the SIB's outgoing branches is taken at runtime, successively determining and building up a coordination sequence.

Coordination sequences are executed by means of a sequence interpreter (*tracer* tool): for each SIB, it delegates the execution to the corresponding execution code, reflecting our policy of separation between coordination and computation: this organization embodies the superposition of the coordination on the components' code and it enables a *uniform view* on the tool functionalities, abstracting from any specific technical details like concrete data formats or invocation modalities.

Inter-component communication is realized via parameter passing and tool functionality invocation: function calls pass as arguments abstract data to the adapters that encapsulate the underlying functionalities The functionalities can be accessed via direct call, *Corba* or *Java RMI* [24] mechanism.

5.5 Model Checking-Based High-Level Validation

Correctness and consistency of the application design is fully automatically enforced in the ABC: throughout the behavior-oriented development process, the ABC offers access to mechanisms for the verification of libraries of constraints by means of model checking. The model checker individually checks hundreds of typically very small and application- and purpose-specific constraints over the flow graph structure. This allows concise and comprehensible diagnostic information in case of a constraint violation since the feedback is provided on the SLG, i.e. at the application level rather than on the code.

The ABC contains an iterative model checker based on the techniques of [20]: it is optimized for dealing with the large numbers of constraints which are characteristic for our approach, in order to allow verification in real time. Concretely, the algorithm verifies whether a given model (a flattened SLG, where the hierarchy information in form of macros has been expended) satisfies properties

expressed in a user friendly, natural language-like macro language [13]. Internally, the logic is mapped to the modal mu-calculus with parameterized atomic propositions and modalities.

- the *properties* express correctness or consistency constraints the target application, e.g., a CTI service, or an internet service, are required to respect.
- the *models* are directly the flattened SLGs, i.e., where macros have been fully expanded, and whereby building block names correspond to atomic propositions, and branching conditions correspond to action names.

Classes of constraints are formed according to the application domain, to the subsystems, and to the purposes they serve. This way it depends on the global goals of an application, which constraints are bound to its model.

Formally, the SLGs are internally interpreted as Kripke Transition Systems [16] whose nodes represent elementary SIBs and whose edges represent branching conditions (see Fig. 4):

Definition 3.
A SLG model *is defined as a triple* $(\mathcal{S}, Act, Trans)$ *where*

- $\mathcal{S}$ *represents the occurrences of SIBs*
- *Act is the set of possible branching condition*
- $Trans = \{(s, a, s')\}$ *is a set of transitions where* $s, s' \in \mathcal{S}$ *and* $a \in Act$.

Global constraints are expressed internally in the *modal mu-calculus* [10]. Internally, we use the following negation-free syntax (positive normal form), which according to De Morgan's law only requires that the set of atomic propositions is closed under negation and that fixpoint variables X in $\mu X.$ or $\nu X.$, appear positively in Φ, i.e. in the range of an even number of negations. Whereas the first assumption is merely technical, the second is standard in order to ensure (the vital) monotonicity of the individual logical operators.

$$\Phi ::= A \mid X \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid [a]\Phi \mid \langle a \rangle \Phi \mid \nu X.\Phi \mid \mu X.\Phi$$

In the above, $a \in Act$, and $X \in Var$, where A is given by the SIB taxonomy, Act by the library of branching conditions, and Var is a set of variables. The fixpoint operators νX and μX bind the occurrences of X in the formula behind the dot in the usual sense. Properties are specified by *closed* formulas, that is formulas that do not contain any free variable.

Formulas are interpreted with respect to a fixed labeled transition system $\langle \mathcal{S}, Act, \rightarrow \rangle$, and an environment $e : Var \rightarrow 2^{\mathcal{S}}$. Formally, the semantics of the mu-calculus is given by:

$$\begin{aligned}
[\![X]\!]e &= e(X) \\
[\![\Phi_1 \vee \Phi_2]\!]e &= [\![\Phi_1]\!]e \cup [\![\Phi_2]\!]e \\
[\![\Phi_1 \wedge \Phi_2]\!]e &= [\![\Phi_1]\!]e \cap [\![\Phi_2]\!]e \\
[\![[a]\Phi]\!]e &= \{\, s \mid \forall s'.\ s \xrightarrow{a} s' \Rightarrow s' \in [\![\Phi]\!]e \,\}
\end{aligned}$$

$$[\![\langle a\rangle \Phi]\!]e = \{\, s \mid \exists s'.\ s \xrightarrow{a} s' \wedge s' \in [\![\Phi]\!]e \,\}$$
$$[\![\nu X.\Phi]\!]e = \bigcup \{S' \subseteq \mathcal{S} \mid S' \subseteq [\![\Phi]\!]e[X \mapsto S']\}$$
$$[\![\mu X.\Phi]\!]e = \bigcap \{S' \subseteq \mathcal{S} \mid S' \supseteq [\![\Phi]\!]e[X \mapsto S']\}$$

Intuitively, the semantic function maps a formula to the set of states for which the formula is "true". Accordingly, a state s satisfies $A \in \mathcal{A}$ if s is in the valuation of A, while s satisfies X if s is an element of the set bound to X in e. The propositional constructs are interpreted in the usual fashion: s satisfies $\Phi_1 \vee \Phi_2$ if it satisfies one of the Φ_i and $\Phi_1 \wedge \Phi_2$ if it satisfies both of them. The constructs $\langle a\rangle$ and $[a]$ are *modal operators*; s satisfies $\langle a\rangle\Phi$ if it has an a-derivative satisfying Φ, while s satisfies $[a]\Phi$ if each of its a-derivatives satisfies Φ. Note that the semantics of $\nu X.\ \Phi$ (and dually of $\mu X.\Phi$) is based on Tarski's fixpoint theorem: its meaning is defined as the greatest (dually, least) fixpoint of a continuous function over the powerset of the set of states.

For the industrial projects it is important to provide a natural language-like feeling for the temporal operators. As indicated by the examples below, the standard logical connectors turned out to be unproblematic. We omit the formal definition of **next**, **generally**, **eventually**, and **until** here and only give the definition of **unless** here, which is used in our example. Please note that in this application scenario, we only use variants of the temporal operators, which *universally* quantify over the set of all possible paths.

$$A \textbf{ unless } B =_{df} \quad \nu X.\ B \ \vee (A \wedge \textbf{next} X)$$

Example: The general OCS policies already mentioned in Sect. 6 and other conference-specific policies inherently define a loose specification of the service at the service logic level, which can be directly formulated as properties of the OCS in our model checking logic. For example, the access control policy is a primary source of constraints like "*A user can modify the defined roles only after having successfully registered as Administrator*", which can be expressed as

$$\neg(\mathsf{modify\text{-}roles}) \textbf{ unless } \ \mathsf{user\text{-}login}\ [\mathsf{Role{=}Admin}]$$

as a global constraint on the SLG of the whole application. This example illustrates the slightly indirect way of expressing the intended constraint. It says,"*A user cannot modify the defined roles unless (s)he has successfully registered as Administrator*". Additionally the example shows a parameterized atomic proposition: user-login [Role=Admin] is parameterized in the possible roles a user might have, and [Role=Admin] does not only require a user-login to appear, but also that the role matches, in this case administrator.

All the properties mentioned earlier in Sect. 3 are requirements expressible in this logic, and they are instances of the classes of safety and consistency requirements identified in [1] to be characteristic of Computer Supported Collaborative Work platforms. They are specific instantiations of Role-Based Access Control (RBAC) models [18]. Being able to automatically verify such properties via model checking is a clear advantage of the ABC, and it is essential in order to guarantee the safety of the kind of applications we build.

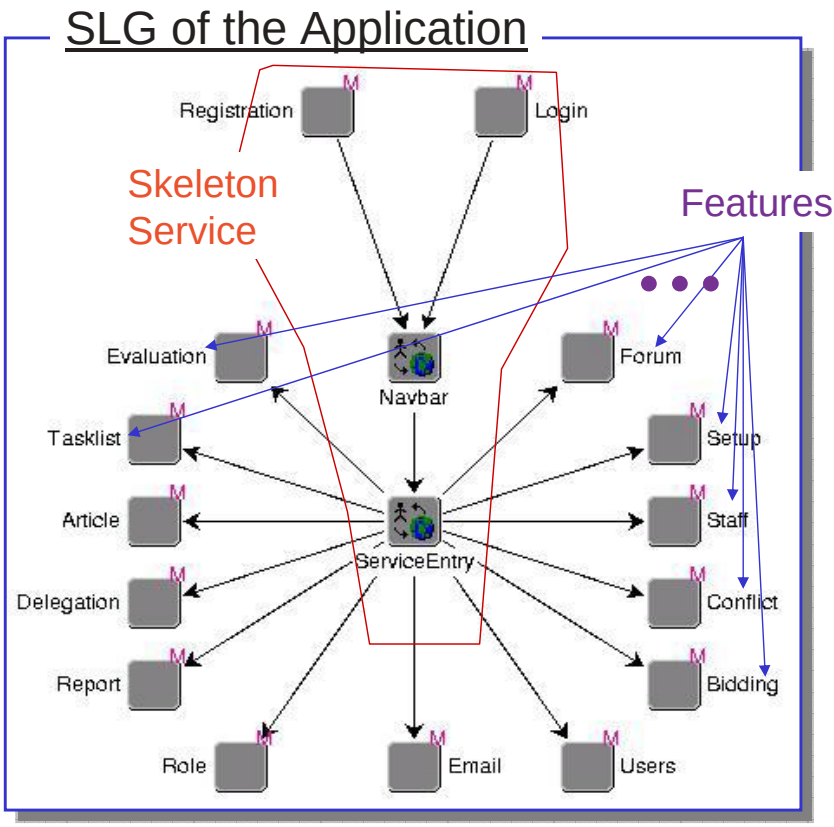


Fig. 5. The Application SLG of the OCS: Skeleton Service and Features

6 The OCS as an Agent and Feature Based System

The user-centered service and application layers in the ABC are structured by means of a hierarchical use of features, which are realized in terms of (possibly nested) macros, each with its own SLG.

Looking at the OCS, a complex application whose SLG has currently approximately 2200 nodes and 3200 edges, we see the typical organization of the different layers in the ABC.

As shown in Fig. 5, the global, application-level SLG is quite simple:

- it contains at the top level the logic of the *backbone service*, which is basically a skeleton service providing generic internet login and session management services, and
- it coordinates the calls to and the interferences between the single *features*.

As we have seen in the feature description, features influence each other, thus one of the aims of service validation, via model checking and testing, is exactly the discovery and control of the so called *feature interactions*.

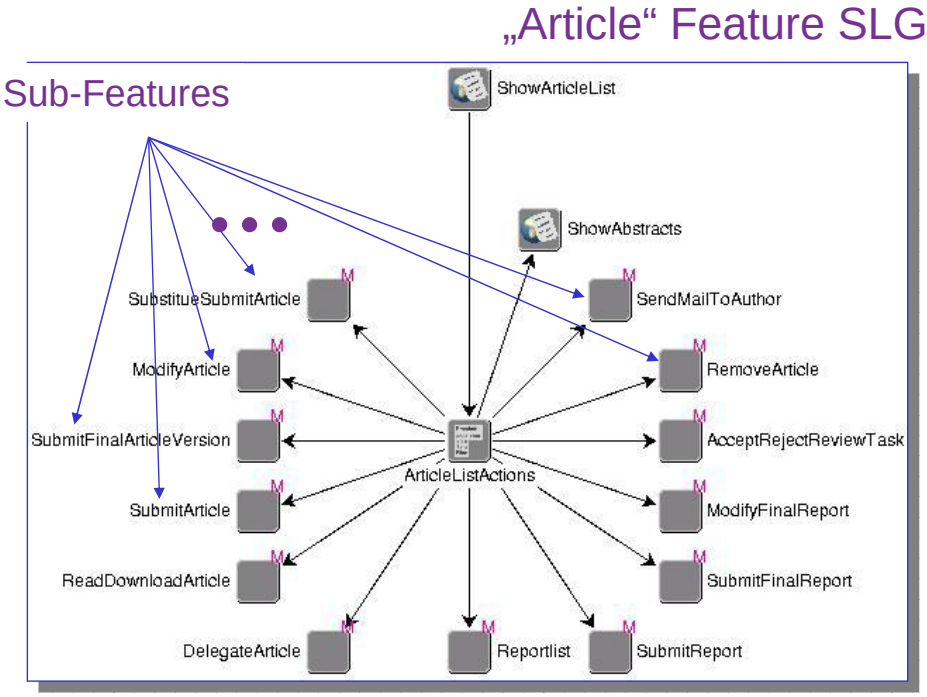


Fig. 6. SLG of the Article Management Feature: Hierarchical Feature Structure

6.1 Features in the ABC

As shown in Fig. 5, each feature is implemented as a macro, thus it has an own Service Logic Graph that defines all the services and the behaviours possible under that feature. Fig. 6 shows, e.g. the SLG that implements the Article Management top-level feature. Top-level features typically provide a number of services to the users. In the case of the OCS, in the depicted version it offers in addition to the Article, Delegation, and Setup Management features already briefly introduced in Sect. 3 also services for the management of Roles, Users, and Staff, as well as e.g. a feature for performing the PC Members' Bidding for papers. This structure becomes immediately evident through the SLG, and it is also explicitly made publicly available over the GUI, as we can see in the navigation bar on the left side of the screen shots of Fig. 1.

New releases of the OCS usually involve the addition or major redesign of top-level features.

6.2 Hierarchy of Features

According to the needs of the application, features can be themselves structured in finer-granular (sub-)features, which are themselves also implemented by means

of SLGs. Similar to the structure at the application level, the SLG of the Article Management feature, shown in Fig. 6,

- contains itself a workflow, here very simple since it provides only navigation capabilities, and
- it coordinates the calls to and the interferences among a number of *finer granular features*, which can be themselves substructured according to the same mechanisms.

In our example, the Article Management feature deals both with the management of articles, as evident from subfeatures like SubmitArticle, ModifyArticle, SubmitFinalArticleVersion, but also with article-related tasks that reside in other features, like Reportlist or DelegateArticle, which are part of the features Role and Delegation respectively.

To illustrate a complete top-down, SLG-based refinement structure, we consider more closely the SubmitArticle subfeature, whose SLG is reported in Fig. 4 and which is technically again implemented as a macro. We reach in this SLG the refinement level where the actual business logic is described, embedded in the context of several checks and of error-handling logic,

1. the ShowSubmitArticle SIB prepares and displays the webpage for the submission action,
2. ShowConfirmArticle allows the user to confirm the submission after checking the correctness of the metadata (like title, article, authors, abstract),
3. then the actual upload in the database and in the CVS versioning system is performed, and finally
4. the ShowSubmitArticleAcknowledgement SIB notifies the submitter of the successful execution.

The SLG also makes use of three macros, CVS_Checkin, mail_notification, and CreateNewsgroup (see Fig. 3, whose SLGs were already used to explain the SIB model and the use of macros. These macros embed reusable pieces of business logic which are relevant to the application designers, but not to the users. Accordingly, these macros do not deserve the status of a feature.

In the ABC, features are enabled and published to the end-users on their finer granularity, according to a complex, personalized role-right-context management. As an example, only users with a PC Chair role are able to submit articles in the name of another user. The design of the sub-structure of features is driven exactly by the needs of distinguishing behaviours according to different contexts. Sub-features in fact usually arise by refinement of features as a consequence of the refinement of the configuration features and of the role-rights management system. This way we enable a very precise fine-tuning of the access to sensitive information and to protected actions.

6.3 Agents in the ABC

Once an internet service is online, it is continuously navigated in parallel by a cohort of agents that execute its global service logic within the limits imposed to the user they are associated with.

Accordingly, in the ABC we do not directly program the agents or their knowledge base. Rather, the SLG of an application defines the space of potential behaviours that agents can assume, and each agent's behaviour is defined implicitly as the currently valid projection onto this potential, filtered via

1. the roles-and-rights management system, which defines dynamic, reconfigurable projections on the behaviours defined in the SLG, and
2. the current global status of the application, including the data space, the configuration, and certain event- and time-dependent permissions.

Technically, the agents developed within the ABC can be classified as

- **residential agents**, in the sense that the service logic resides on a single machine (more precisely on a load-balanced cluster, which is however transparent to the SLG and therefore to the agent), which are
- **proactive**, in the sense that the agent *knows autonomously* which features are enabled at this moment for his user, which contents are available for him, and which consequences has a user action wrt. the further navigation potential within this session, and
- **online adaptive**, in the sense that they react instantly to changes in the context (user preferences, configurations of role and rights, other actions by other agents). E.g., a report checked in by the last reviewer may automatically open a discussion forum for this paper available for the whole PC.

7 Conclusions

In this paper, we have shown how the concepts of objects, components, features, and agents are used today in the *Application Building Center* (ABC) environment in order to marry the modelling of functionally complex communication systems at the application level with an object-oriented, component based implementation. We are not aware of any approach which is similar in its intent, which is in particular characterized by simplicity of the modelling level. The closest approaches we know of typically require far more knowledge at the application level (at least programming expertise) and/or lack systematic support by means of formal methods, and therefore are inadequate for the scenarios and users we address.

The impact of this approach on the efficiency of the design and documentation has been proven dramatic in industrial application scenarios: our industrial partners reported a performance gain in time-to-market of the applications of a factor between 3 and 5. The reason for the reported gain was mainly the early error detection, due to the tightened involvement of the application expert into the development cycle. More generally, we see the current approach as an instance of Model Driven Application Development, where heterogeneous models allow the individual, but interdependent modelling of complementary aspects. And indeed, objects, components, features, and agents constitute specific categories of such aspects, adequate for the structuring of complex applications according

to complementary views. Even though it is only a very first step, we consider the ABC a kind of proof of concept motivating the design of more advanced model-based techniques, in particular concerning elegant and powerful approaches to modelling and proving compatibility.

Acknowledgements

Many thanks are due to Martin Karusseit for the support and help in discussions on the OCS, and to Bernhard Steffen for his constructive critique on previous versions of the paper.

References

1. T. Ahmed, A. Tripathi: *Static Verification of Security Requirements in Role Based CSCW Systems*, Proc. 8th Symp. on Access Control Models and Technologies, Como (I), ACM Press, pp.196-203, 2003.
2. L. Andrade, J. Fiadeiro, J. Gouveia, G. Koutsoukos, A. Lopes, M. Wermelinger: *Coordination Technologies For Component-Based Systems*, Int. Conference on Integrated Design and Process Technology, IDPT-2002, Pasadena (CA), June, 2002, Society for Design and Process Science.
3. J. Bredereke: *On Feature Orientation and Requirements Encapsulation*, this vol.
4. *CVS: Concurrent Versions System*, www.cvshome.org/
5. A. Hagerer, T. Margaria, O. Niese, B. Steffen, G. Brune, H.-D. Ide: *An Efficient Regression Testing of CTI Systems: Testing a complex Call-Center Solution*, Annual Review of Communic., Vol. 55, Int. Engineering Consortium, Chicago, 2001.
6. H. Harris, M. Ryan: *Theoretical Foundations of Updating Systems*. ASE 2003, 18th IEEE Int. Conf. on Automated Software Engineering, IEEE-CS Press, 2003.
7. ITU: *General recommendations on telephone switching and signaling – intelligent network: Introduction to intelligent network capability set 1*, Recommendation Q.1211, Telecommunication Standardization Sector of ITU, Geneva, Mar. 1993.
8. ITU-T: *Recommendation Q.1203. "Intelligent Network - Global Functional Plane Architecture"*, Oct. 1992.
9. ITU-T: *Recommendation Q.1204. "Distributed Functional Plane for Intelligent Network Capability Set 2: Parts 1-4"*, Sept. 1997.
10. D. Kozen: *Results on the Propositional μ-Calculus*, Theoretical Computer Science, Vol. 27, 1983, pp. 333-354.
11. B. Lindner, T. Margaria, B. Steffen: *Ein personalisierter Internetdienst für wissenschaftliche Begutachtungsprozesse*, GI-VOI-BITKOM-OCG-TeleTrusT Konferenz Elektronische Geschäfts-prozesse (eBusiness Processes), Universität Klagenfurt, September 2001, http://syssec.uni-klu.ac.at/EBP2001/ .
12. T. Margaria, M. Karusseit: *Community Usage of the Online Conference Service: an Experience Report from three CS Conferences*, 2nd IFIP Conf. on "e-commerce, e-business, e-government" (I3E 2002), Lisboa (P), Oct. 2002, in "Towards the Knowledge Society", Kluwer, pp.497-511.
13. T. Margaria, B. Steffen: *Lightweight Coarse-grained Coordination: A Scalable System-Level Approach*, to appear in STTT, Int. Journal on Software Tools for Technology Transfer, Springer-Verlag, 2003.

14. T. Margaria, M. Wuebben: *Tool Integration in the ETI Platform - Review and Perspectives*, TIS 2003: ESEC/FSE Workshop on Tool Integration in System Development, Sept. 2003, Helsinki (FIN), pp.39-44
15. T. Margaria, B. Steffen: *Aggressive Model-Driven Development: Synthesizing Systems from Models viewed as Constraints* (invited position paper) Monterey'03 Worksh. on "Software Engineering for Embedded Systems: From Requirements to Implementation", Chicago, Sept. 2003, - Post workshop Proceedings in print, IEEE-CS Press.
16. M. Müller-Olm, D. Schmidt, B. Steffen: *Model-Checking: A Tutorial Introduction*, Proc. SAS'99, September 1999, LNCS 1503, pp. 330–354, Springer Verlag.
17. Object Management Group: *The Common Object Request Broker: Architecture and Specification*, Revision 2.3, Object Management Group, 1999.
18. R. Sandhu, E. Coyne, H. Feinstein, C. Youman: *Role-Based Access Control Models*, IEEE Computer, 29(2):38-47, Feb. 1996.
19. W3C: SOAP *http://www.w3.org/TR/SOAP/*
20. B. Steffen, A. Claßen, M. Klein, J. Knoop, T. Margaria: *The Fixpoint Analysis Machine*, (invited paper) CONCUR'95, Pittsburgh (USA), August 1995, LNCS 962, Springer Verlag.
21. B. Steffen, T. Margaria: *METAFrame in Practice: Intelligent Network Service Design*, In *Correct System Design – Issues, Methods and Perspectives*, LNCS 1710, Springer Verlag, 1999, pp. 390-415.
22. B. Steffen, T. Margaria, V. Braun: *The Electronic Tool Integration platform: concepts and design*, Int. J. STTT (1997)1, Springer Verlag, 1997, pp. 9-30.
23. B. Steffen, T. Margaria, V. Braun, N. Kalt: Hierarchical service definition, Annual Review of Communication, Int. Engin. Consortium (IEC), 1997, pp. 847-856.
24. Sun: *Java Remote Method Invocation*. `http://java.sun.com/products/jdk/rmi`.
25. H. Weber: *Continuous Engineering of Information and Communication Infrastructures*, Proc. Int. Conf. on Fundamental Approaches to Software Engineering (FASE'99), Amsterdam, LNCS N. 1577, Springer Verlag, pp. 22-29.

Towards a Formal Specification for the AgentComponent

Philipp Meier and Martin Wirsing

Ludwig-Maximilians-Universtität München
{meierp,wirsing}@informatik.uni-muenchen.de

Abstract. In this paper we present the AgentComponent (AC) approach under formal aspects. A multi agent system (MAS) is composed of AC instances, where each AC instance consists of a knowledge base, storing the beliefs, of slots, storing the communication partners, of a set of ontologies, and of so-called ProcessComponents (PC) representing the behaviours of an AC instance. An AC is a generic component that can be reused (instantiated ACs) and parametrized by customizing the communication partners (slots), the ontologies, the knowledge and the behaviours of an AC instance. The focus of this paper is to introduce the basic AC approach and to define a formal specification using the Object Constraint Language (OCL) for this approach. In the first part we present an informal description of the AC approach and show how to construct a component-based MAS by a simple example. The second part of this paper presents a metamodel for the AC approach and provides a formal specification for the services of the generic AC.

1 Introduction

Software agents and software components are two emerging fields in software development [8]. Each concept describes different ways for building software but both concepts claim to improve the way large scaled software is built. While components focus more on reusability aspects of software, agents (MAS) focus on processing complex tasks collaboratively.

Combining Agents and Components. Our idea is based on combining both concepts to integrate the advantages of component technology into agent technology. For this purpose many similarities can be found between agent and component technology [6]. But we are more interested in the complementary concepts of these technologies, because these are the key concepts that make the profit of combining agents and components. We see the main purpose of agents in the communication ability, which gives one the possibility to process complex tasks by assigning single tasks to different agents. Instead components focus more on reusability and parametrisation/customization aspects for the deployment of a component in different contexts [8, 7]. In our opinion there exist two different ways of combining agent and component technology. The first one, we call it "agentifying", sees component technology as starting point and tries to include agent properties into existing components. The second one, we

M.D. Ryan et al. (Eds.): Objects, Agents, and Features, LNCS 2975, pp. 175–188, 2004.

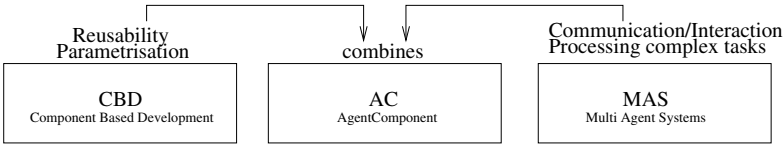


Fig. 1. Combination Profit

call it "componentifying", considers agent technology as starting point and tries to add component features to existing agent technology. Our approach which is based upon the "componentifying concept" combines the main features of agents and components in the so-called AgentComponent (AC). In this way we achieve both: An AC that has communication abilities (the agent as starting point) and can be reused and parametrized for different contexts (see Fig. 1). With this approach we try to disburden agent software development and we found component technology as a suitable instrument for this task by making certain agent concepts customizable and reusable. Here "reusable" means to have an AC that can be instantiated for every agent we need and "customizable" means to connect AC instances, add/remove certain ontologies and add/remove behaviours (graphically). Moreover we consider these customization services of the AC under formal aspects. We use signatures, UML class diagrams and the Object Constraint Language (OCL) [9] to specify the subcomponents and services of the generic AC.

In section 2 we describe the basic concepts of the AC approach, where we give a simple example to illustrate these concepts. Section 3 and 4 present the metamodel for the AC approach and give a formal specification of the services of the AC. Finally, section 5 describes some related work and section 6 concludes the paper.

2 AgentComponent

In general, an AC instance has all agent properties like autonomy, reactivity, proactivity and interaction and integrates component features like reusability and customizability (according to the "componentifying concept"). An AC is a generic component that describes general services that every agent must provide in any context. So an AC can be instantiated/reused for every agent one wants to build [10, 11]. A user can easily customize an AC instance for his purpose and for his context in the following ways:

(1) Customizing the ACs' communication partners. We introduce so-called *slots* that hold information about communication partners of AC instances. Communication partners can be added to or removed from the slots of AC instances (customization of slots).
(2) Customizing the ACs' ontologies. Ontologies can be registered or deregistered from AC instances.

(3) Customizing the ACs' processes. We introduce so-called *ProcessComponents* (PCs) that implement and describe simple or complex behaviours that can be added to or removed from any AC instance.

With these three facilities the *appearance* and the *behaviour* of AC instances can be customized. Therefore every AgentComponent includes the following entities for which it has to provide the following services: A knowledge base, slots for communication partners (holding communication partners), a slot for ontologies and a slot for ProcessComponents:

Knowledge Base. The knowledge base is a basic element of every AC instance. An AC provides services to add, remove and get resources from the knowledge base. The following services are provided by an AC for the knowledge base concept:

(1) *retrieve* (getting knowledge from the knowledge base).
(2) *assert* (adding knowledge to the knowledge base).
(3) *retract* (removing knowledge from the knowledge base).

Ontologies. Ontologies are basic agent concepts that agents use to understand the content of the messages that are being sent or received. So for certain contexts ontologies are provided that can be registered with AC instances. Instances of these ontologies can then be used as common language between AC instances and can also be asserted to, retracted or retrieved from the knowledge base. The following services are provided by the AC for the ontologies concept:

(1) *addOnto* (register an ontology with an AC instance).
(2) *removeOnto* (deregister an ontology with an AC instance).

Communication Slots. Communication slots are entities of AC instances that hold information about communication partners (other agents). Communication pathways between AC instances can be created w.r.t. customized by connecting and changing slots, in this way filling the slots with the required destinations (other AC instances) for communication. Using these slots messages can be sent and received via the created pathways. The following services are provided by the AC for the slot concept:

(1) *addSlot* (adds a slot for communication partners to AC instances).
(2) *removeSlot* (removes a slot from AC instances).
(3) *connectSlot* (fills a slot with destinations).
(4) *disconnectSlot* (removes a destination from a slot).

ProcessComponents. A ProcessComponent represents a certain behaviour that can be added and removed to an AC instance.

(1) *addPC* (adds a PC to an AC instance).
(2) *removePC* (removes a PC from an AC instance).

When a PC has been added to an AC instance the PC has full access to all services that the AC provides (see Fig. 2). I.e. the PC is able to assert, retract, retrieve resources, to send and receive messages and moreover to use all the services listed in the StructuralInterface (see Fig. 2). So every PC implements an activity graph that consists of assert-, retract-, retrieve-, send-, receive-,

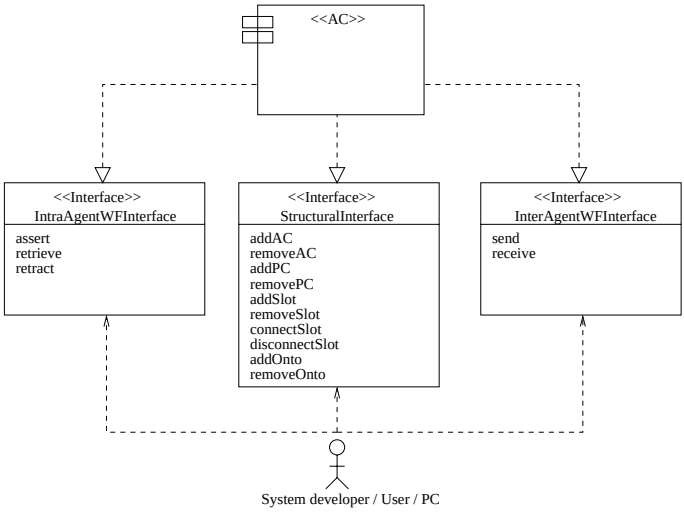


Fig. 2. AgentComponent Services

StructuralInterface- or custom activities. In this way a PC defines which input it requires (retrieve, receive) and the output it delivers (assert, retract, send, StructuralInterface services). By adding or removing PCs from an AC instance a system developer or user can determine the *appearance* (the structure of an AC instance, see StructuralInterface in Fig. 2) and the *behaviour* of an AC. Most important a PC determines the behaviour of an AC instance, because a PC implements a certain workflow, represented by an activity graph. So the PCs fill each AC instances with life by defining and implementing certain tasks for agents. But a PC has also the ability to determine the appearance of an AC instance. We have mentioned above that a PC can use all the services that are included in the StructuralInterface. All these services define the *appearance* of an AC instance and are ment to customize the structure of an AC instance. On the one hand we want the user or system developer to do this structural customization (graphically) and on the other hand the PCs can implement behaviours (emergent bevahiours) that customize and determine the structure of an AC instance. Therefore customization can be done in both ways, by the user (graphically) and by PCs.

Intra-agent and Inter-agent Workflow. Up to know we have not talked about how AC instances or better their PCs can work together and define a workflow. This is important to explain, because we do not provide a workflow control within an AC (as e.g. [5] does) and leave the workflow and its control to the PC implementor. The PCs themselves are supposed to work autonomously, i.e. PCs describe only their own workflow by defining what they require (receive, retrieve) and what they deliver (send, assert, retract, StructuralInterface services) not caring about the overall workflow. The way a workflow between

two or more PCs is handled must be divided into two concepts, the inter-agent and the intra-agent workflow. Obviously the *inter-agent* workflow between AC instances is handled by sending and receiving messages using the pathways between AC instances. These pathways are stored in the slots and can be changed using the "connectSlot" or "disonnectSlot" service. Whereas the assert, retract and retrieve services handle the *intra-agent* workflow (workflow within an AC), by changing the knowledge base and therefore making resources available for other PCs (of the same AC instance) that wait for these resources. In this way we use the knowledge base as a "shared memory" for input and output variables, that can be used by all PCs added to an AC instance. The following services must be provided by an AC for the inter-agent workflow concept:

(1) *send* (sending messages to other AC instances).
(2) *receive* (receiving messages from other AC instances).

Putting all the mentioned concepts together the PC implementor has to provide:

(1) The implementation of the PC, where all services of the AC can be used to send, receive messages, to retrieve, assert and retract resources from the knowledge base and to customize the structure.

(2) The PC description, where one must provide: (2.1) The description of all ontologies, that are required by the PC,
(2.2) the information about required slots for the inter-agent communication and (2.3) the description of the workflow of the PC (mainly the description of the usage of the services), so that new PC providers understand the activities w.r.t. the behaviour of the PC.

Projects and AgentGroups. Agents are loosely coupled entities by nature, that communicate using messages and protocols and that are not directly aware of other agents around them (only indirectly via the AMS or the DF). We introduce the term *Project* as a special case of a PC. A Project normally describes an unspecified task, e.g. an AC instance does not know which other ACs are able to assist in collaboratively executing a task. So this AC has to look for one or more ACs that can handle this task. After this search they will join for this special task at runtime and split up after finishing the Project. Projects are normally described by 1:n protocols and have a highly dynamic character. What misses in this highly dynamic environment is the fact that software systems only in some cases need such dynamic character. So, beside the Projects, we see the necessity to be able to connect the ACs directly using the before introduced slots. So we are able to build up an acquaintance net (AgentGroup) with an organization structure according to the pathways described in the slots. System developers or users will be able to create AgentGroups (graphically) using a tool, whereas Projects normally construct their organization structure and information pathways during runtime. The following services must be additionally provided by an AC for the AgentGroup concept:

(1) *addAC* (adds an AC to the AgentGroup of an existing AC instance).
(2) *removeAC* (removes an AC instance from an AgentGroup).

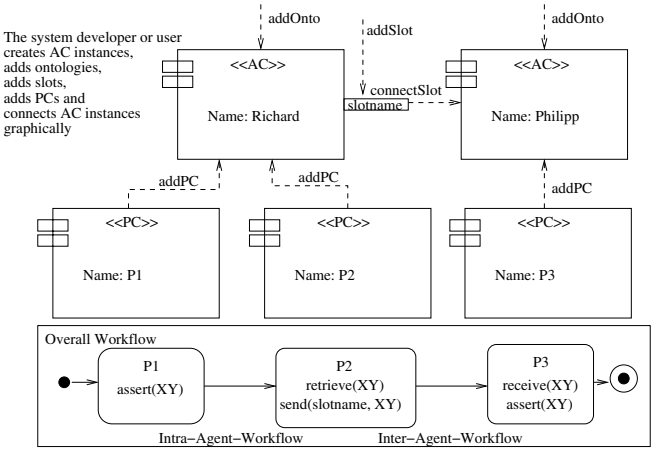


Fig. 3. Constructing a Simple AgentGroup

Example. To illustrate these basic concepts of the AC approach we describe a very small example. In this example (see Fig. 3) we describe the construction of a simple system, assuming that there exists a tool that supports the construction graphically. Furtheron, we assume that three PCs and their information description about ontologies, slots and behaviours are available. So we can use the three PCs called P1 to P3. P1 asserts an ontology object named XY to the knowledge base. P2 retrieves an XY ontology object and sends it to any AC instance available (we do not know to which AC instance we want to send so far, depends on the entries in the used slot). P3 receives a message containing an XY ontology object and asserts it to the ACs' knowledge base. P2 and P3 provide a description for the ontologies and for the slots they need during their workflow. First of all we create two idle AC instances named Richard and Philipp. Before we add any PC to Richard or Philipp we need to customize the structure of the both AC instances to make the intra-agent and inter-agent workflow of the PCs working. I.e. as we want to add P1 and P2 to Richard and P3 to Philipp, Richard and Philipp need the slots and the ontologies according to the PC descriptions. After creating a slot it can be filled with any destinations available, in our case we connect it with Philipp, and hereby Philipp is stored as a destination in a slot of Richard. By adding all this structural information and adding the PCs as mentioned before we define on the one hand an intra-agent workflow where P2 requires the output (retrieve(XY)) of P1 (assert(XY)). And on the other hand we define an inter-agent workflow between P2 (send(XY)) added to Richard and P3 (receive(XY)) added to Philipp. After adding the PCs their behaviours start immediately and the workflow is executed (see Fig. 3).

Summary. We have shown an AC that provides three interfaces (InterAgentWorkflowInterface, IntraAgentWorkflowInterface and StructuralInterface) including common services every agent requires. System developers do not have to implement the AC, they just need to instantiate the AC as often as they

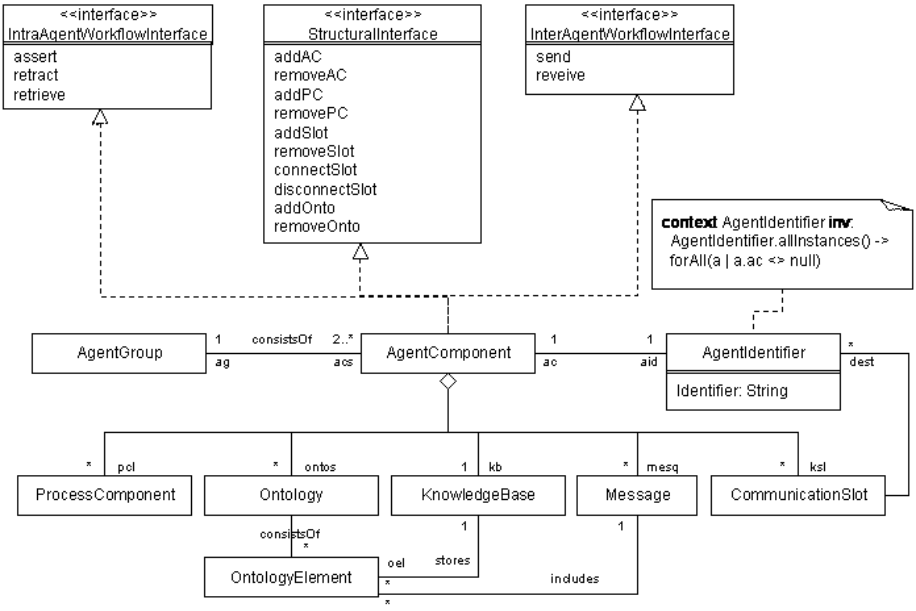


Fig. 4. AgentComponent Metamodel

require it. To fill the ACs with life PCs must be provided by the developers which then can be added to the AC instances. This results in an ACS (Agent Component System - assembled of ACs) which uses customized inter-agent communication (achieved by the concept of slots and ontologies) and uses customized intra-agent communication (achieved by the concept of PCs using the knowledge base).

3 Metamodel for the AgentComponent Approach

Putting together all the informally described AC concepts in the former sections, we present now a metamodel for the AC approach. Fig. 4 shows the whole metamodel in a UML class diagram. According to this class diagram an AgentComponent is composed of a set of ProcessComponents, a set of Ontologies, a KnowledgeBase and a set of CommunicationSlots. Moreover every AgentComponent holds a queue of received messages. An Ontology is defined by OntologyElements which define the concepts for certain contexts, and these OntologyElements can be stored in the KnowledgeBase. CommunicationSlots store no direct references to the AgentComponents, instead they hold AgentIdentifiers that represent the correspondent AgentComponents. Therefore each AgentComponent is identified by an unique AgentIdentifier and is a member of an AgentGroup. Moreover the AgentIdentifier invariant defines, that for every AgentIdentifier a corresponding AgentComponent must exist. In this way we assure that an AgentIdentifier that can be stored in a CommunicationSlot must have a corresponding AgentCom-

ponent. If an AgentIdentifier would exist without a correspondent AgentComponent, Messages sent to this AgentIdentifier could not be received because the correspondent AgentComponent for this AgentIdentifier does not exist. For customization of these building blocks an AgentComponent implements the services shown in the three interfaces (introduced before). In this way users and ProcessComponents have access to the building blocks of an AgentComponent using these services (as shown in Fig. 2). In the next section we present the formal specification of all the services an AgentComponent provides.

4 Formal Specification of the AgentComponent Services

The specification of the generic AC is processed in two steps. First, we give a signature for the AC where we define the parameters and return values for the services of the AC. Second, we specify the servies of the AC using OCL [9] and define pre- and post-conditions for every service. OCL as a part of UML is normally used to specify object-oriented systems. OCL constrains a object system in two ways: Using invariants for class diagrams all possible systems states are restricted to the ones we want to allow. In this way invariants can be used to constrain the static aspects of object systems. The dynamic aspects of an object system are restricted by pre- and post-conditions for methods. Using these pre- and post- conditions we can restrict the system state transitions to the ones we want to allow. We will use this notion to specify the dynamic aspects of our generic AgentComponent. So we will specify pre- and post-conditions for all AC services described in the above chapters.

4.1 Signature for the AgentComponent Services

First of all we define a signature for the AgentComponent based on the metamodel of Fig. 4. We define a signature for a class diagram as follows.

Definition 1. *A signature $\Sigma_\Delta = (S, OP)$ for a class diagram Δ consists of*

- *a set S of sorts, also called types (names for data domains)*
- *a set OP of operations (services of the AC) of the form $op : s_1 \times \ldots \times s_n \rightarrow s$ with $s_1, \ldots, s_n, s \in S$ $(n \geq 0)$*

According to this definition we define the signature Σ_Δ for the metamodel class diagram as follows. The services of the AC are shown with their parameters and return values.

$$
\begin{aligned}
S = \{ & \mathit{AgentComponent}, \mathit{AgentGroup}, \mathit{AgentIdentifier}, \\
& \mathit{ProcessComponent}, \mathit{Ontology}, \mathit{KnowledgeBase}, \mathit{Message}, \\
& \mathit{CommunicationSlot}, \mathit{OntologyElement}, \mathit{Collection}(\mathit{ProcessComponent}), \\
& \mathit{Collection}(\mathit{Ontology}), \mathit{Collection}(\mathit{Message}), \\
& \mathit{Collection}(\mathit{CommunicationSlot}), \mathit{Collection}(\mathit{OntologyElement}) \}
\end{aligned}
$$

$$
\begin{aligned}
OP = \{ & assert : AgentComponent \times OntologyElement \rightarrow Void, \\
& retract : AgentComponent \times OntologyElement \rightarrow Void, \\
& retrieve : AgentComponent \times OntologyElement \rightarrow \\
& \qquad Collection(OntologyElement), \\
& send : AgentComponent \times Message \times String \rightarrow Void, \\
& receive : AgentComponent \times Ontology \rightarrow Collection(Message), \\
& addAC : AgentComponent \times AgentComponent \rightarrow Void, \\
& removeAC : AgentComponent \times AgentComponent \rightarrow Void, \\
& addSlot : AgentComponent \times String \rightarrow Void, \\
& removeSlot : AgentComponent \times String \rightarrow Void, \\
& connectSlot : AgentComponent \times String \times AgentIdentifier \rightarrow Void, \\
& disconnectSlot : AgentComponent \times String \times AgentIdentifier \rightarrow Void, \\
& addOnto : AgentComponent \times Ontology \rightarrow Void, \\
& removeOnto : AgentComponent \times Ontology \rightarrow Void, \\
& addPC : AgentComponent \times ProcessComponent \rightarrow Void, \\
& removePC : AgentComponent \times ProcessComponent \rightarrow Void\}
\end{aligned}
$$

4.2 Specification of the Services of the AgentComponent

Based on the signature definition and the class diagram in Fig. 4 we give now a formal specification of the AC services (OP in the signature definition) using the Object Constraint Language (OCL) [9]. We assume that the following methods are given, so they can be used in the pre- and post-conditions without further specification.

- $getCommunicationSlot : AgentComponent \times String \rightarrow CommunicationSlot$: This method searches through the set of Communication slots (ksl) by a given String and returns the found CommunicationSlot.
- $getType : OclAny \rightarrow OclType$: This method returns the type (class) of an object.
- $getOnto : Message \rightarrow Ontology$: This method returns the ontology of an message.

The IntraAgentWFInterface Services

```
context AgentComponent::assert(oe: OntologyElement): Void
  pre : oe <> null
  post: kb.oel = kb.oel@pre -> including(oe)

context AgentComponent::retract(oe: OntologyElement): Void
   pre: oe <> null
  post: if kb.oel@pre -> includes(oe) then
          kb.oel = kb.oel@pre -> excluding(oe)
        else
          kb.oel = kb.oel@pre
        endif
```

```
context AgentComponent::retrieve(oe: OntologyElement): Collection
   pre: oe <> null
  post: result = kb.oel -> select(o |
          o.oclIsKindOf(oe.getType()))
```

Each of the services test in their pre-condition whether the given parameter does not equal null. The post-condition of the assert service assures that the given OntologyElement is now in the KnowledgeBase. The post-condition of retract assures that if the OntologyElement has been stored in the KnowledgeBase before, that it is now removed from the KnowledgeBase. Finally the retrieve service looks for all OntologyElements stored in the Knowledgebase that are equal to the type of the given OntologyElement parameter and returns the results in a collection.

The InterAgentWFInterface Services

```
context AgentComponent::send(msg: Message,csName : String): Void
  pre : csName <> null
  post: if getCommunicationSlot(csName) <> null then
          getCommunicationSlot(csName).dest ->
          forAll(ai | ai.ac.mesq = ai.ac.mesq@pre ->
          including(msg))
        endif

context AgentComponent::receive(onto: Ontology): Collection
pre:    ontos -> includes(onto)
post:   result = mesq@pre -> select(m | m.getOnto() = onto)
post:   mesq = mesq@pre -> reject(m | m.getOnto() = onto)
```

The send service takes a message and the name (csName) of the slot where the message should be sent to. The post-condition of this service assures that the sent message is stored in all message queues of the AC instances that are stored in the slot "csName" and therefore received the message. The receive service takes an Ontology as parameter, searches for Messages in the message queue which contain OntologyElements of the given Ontology (onto) and returns the results in a collection.

The StructuralInterface Services

```
context AgentComponent::addAC(ac: AgentComponent): Void
  pre : ac <> null
  post: ag.acs = ag.acs@pre -> including(ac)

context AgentComponent::removeAC(ac: AgentComponent): Void
   pre: ag.acs -> includes(ac)
  post: ag.acs = ag.acs@pre -> excluding(ac)
  post: AgentComponent.allInstances() -> forAll(a | a.ksl ->
        forAll(cs | cs.dest -> forAll(ai |
```

```
        if ac.aid = ai
        then cs.dest = cs.dest@pre -> excluding(ai) endif )))
```

The post-condition of the addAC service simply assures whether the given AgentComponent instance (ac) is included in the AgentGroup. The removeAC service first has to test if the given AgentComponent instance (ac) is included in the AgentGroup. The post-condition of this service assures that the AgentComponent instance is removed from the AgentGroup and that the AgentIdentifier of the removed AC is removed from all existing slots of the other AC instances.

```
context AgentComponent::addSlot(csName: String): Void
   pre: csName <> null
  post: if getCommunicationSlot(csName) = null then
          ksl = ksl@pre -> including(getCommunicationSlot(csName))
        else
          ksl = ksl@pre
        endif

context AgentComponent::removeSlot(csName: String): Void
   pre: csName <> null
  post: if getCommunicationSlot(csName) <> null then
          ksl = ksl@pre -> excluding(getCommunicationSlot(csName))
        endif

context AgentComponent::connectSlot(csName: String,
                                    ai: AgentIdentifier): Void
   pre: csName <> null and
        getCommunicationSlot(csName) <> null
  post: getCommunicationSlot(csName).dest =
        getCommunicationSlot(csName).dest@pre -> including(ai)

context AgentComponent::disconnectSlot(csName: String,
                                       ai: AgentIdentifier): Void
   pre: csName <> null and
        getCommunicationSlot(csName) <> null and
        getCommunicationSlot(csName).dest -> includes(ai)
  post: getCommunicationSlot(csName).dest =
        getCommunicationSlot(csName).dest@pre -> excluding(ai)
```

The addSlot w.r.t the removeSlot assure that an CommunicationSlot with the given name (csName) is added w.r.t removed from the set of slots. The connectSlot and disconnectSlot services take as an additional parameter an AgentIdentifier (ai) and assure that the given slot is added to or removed from the given slot(csName).

```
context AgentComponent::addOnto(onto: Ontology): Void
   pre: onto <> null
```

```
  post: if ontos -> excludes(onto) then
          ontos = ontos@pre -> including(onto)
        else
          ontos = ontos@pre
        endif

context AgentComponent::removeOnto(onto: Ontology): Void
   pre: onto <> null
  post: if ontos -> includes(onto) then
          ontos = ontos@pre -> excluding(onto)
        else
          ontos = ontos@pre
        endif
```

The addOnto and removeOnto services take an Ontology as parameter and assure that this Ontology (onto) is added to w.r.t. removed from the collection of Ontologies (ontos).

```
context AgentComponent::addPC(pc: ProcessComponent): Void
   pre: pc <> null
  post: pcl = pcl@pre -> including(pc)

context AgentComponent::removePC(pc: ProcessComponent): Void
   pre: pc <> null
  post: pcl = pcl@pre -> excluding(pc)
```

The addPC and removePC services take a ProcessComponent as parameter and assure that it is added w.r.t. removed from the ProcessComponent collection (pcl)

5 Related Work

There exist some work on component-based agent technology [4, 5, 12, 1, 2]. All these approaches define different (sub)components an agent is composed of. But taking a closer look to these approaches their component definition does not match with the commonly used component definitions as proposed by [3] or by the Object Management Group (OMG). According to this definition the components' interal structure is considered as a "blackbox", that is encapsulated by the components' interfaces. The crucial point here is to define the services of a component that it provides w.r.t. requires. With this specification the functionality of an component is defined and the component can be accessed from outside. We missed exactly this specification of the functionality (interfaces) in the other approaches. Moreover we proposed a generic AC that can be reused as a whole by simply instantiating the generic AC and customizing it for different contextes using the specified services [10, 11].

DESIRE [5] describes a component-based design method for agents, where the agent is split up in process components and knowledge components. [5]

also provides component templates for different agent types like weak agents, strong agents or BDI-agents. In contrast to DESIRE we have no compositional processes and knowledge, instead we propose autonomously working processes (PCs). Moreover our approach provides workflow services but no workflow control for the processes as proposed by [5]. This gives us more flexibility in exchanging the behaviours of an agent.

FIPA-OS [4] is a component-oriented multi-agent-framework for constructing FIPA compliant Agents using mandatory components (i.e. components required by ALL FIPA-OS Agents to execute) and optional components (i.e. components that a FIPA-OS Agent can optionally use). The mandatory parts of an FIPA-OS-Agent are the Agent class which can be extended, the TaskManager that represents the functionality of an agent, the Conversation Manager for protocol-like conversation between agents and the Message Transport Service for sending and receiving messages. ZEUS, agentTool and AGIL [12, 1, 2] are visual agent toolkits for building MAS. Each agent can be visually constructed from different components (e.g. ZEUS uses Ontologies, Tasks, organization structures to represent an agent; AGIL builds their agents from workflow activities). For these visually designed components the correspondent agent code is generated (mostly Java code) and the whole MAS is deployed on a platform. The main difference between these tools and the AgentComponent tool [10, 11] is that the AC Tool is not a code generator for agent systems. It is build for visually constructing agent systems and customizing these systems during runtime according to the AgentComponent concepts proposed in this paper. So an AgentComponent system never needs to be shut down to change agent organisation, agent behaviour or agent knowledge.

6 Concluding Remarks

The AgentComponent approach integrates component aspects into agent technology in order to benefit from the advantages of each of the both technologies. According to the "Componentifying" concept we have included the main component concepts into agent technology by making certain agent concepts customizable (see Sect. 2) and reusable (AgentComponent). For this purpose we have defined the AC as a component that provides certain services (the interfaces of the AC) for every agent. We have shown the metamodel (signature) for the AC approach and, based on this model, we have specified the interface services of the AC.

In this way we can instantiate and reuse the AC for every agent we want to build and instantiated ACs can be customized in the following ways:

1. customizing the structural elements of the AC instance (achieved by the services of the StructuralInterface),
2. customizing the inter-agent communication (achieved by the concept of slots) and
3. customizing the intra-agent communication (achieved by the concept of PCs using the knowledge base).

References

1. agentTool Website. http://www.cis.ksu.edu/~sdeloach/ai/agentool.htm.
2. AGIL Website. http://www1.faw.uni-ulm.de/kbeans/agil/.
3. Desmond Francis D'Souza, Alan Cameron Wills. *Objects, Components and Frameworks With UML*. Addison Wesley, 1999.
4. FIPA-OS Website. http://www.emorphia.com/research/about.htm.
5. Frances M.T. Brazier, Catholijn Jonker, Jan Treur. Principles of Component-Based Design of Intelligent Agents. *Data and Knowledge Engineering*, 41:1 – 28, 1998.
6. Juergen Lind. Relating Agent Technology and Component Models, 2001.
7. M. Wooldridge and P.Ciancarini. Agent-Oriented Software Engineering: The State of the Art. *Agent-Oriented Software Engineering. Springer-Verlag Lecture Notes*, 1957, 2001.
8. Nick R. Jennings et al. Agent-Based Business Process Management. *ACM SIGMOD Record 27*, pages 32–39, 1998.
9. Object Constraint Language Specification. http://www.cvc.uab.es/shared/teach/a21291/apunts/OCL/apunts/OCL-Specifi%cation.pdf.
10. Philipp Meier. Visual Construction of Multi-Agent-Systems according to the AgentComponent Approach and the RDT Concept, Technical Report. http://www.pst.informatik.uni-muenchen.de/publications/agentcomponenttool.pdf.
11. Philipp Meier, Martin Wirsing. Implementation Patterns for Visual Construction of Multi-Agent-Systems, Technical Report. http://www.pst.informatik.uni-muenchen.de/publications/acpatterns.pdf.
12. Zeus Website. http://193.113.209.147/projects/agents.htm.

Policies: Giving Users Control over Calls

Stephan Reiff-Marganiec

University of Leicester*,
Department of Computer Science,
Leicester LE1 7RH, UK
srm13@le.ac.uk

Abstract. Features provide extensions to a basic service, but in new systems users require much greater flexibility oriented towards their needs. Traditional features do not easily allow for this. We propose policies as the features of the future. Policies can be defined by the end-user, and allow for the use of rich context information when controlling calls. This paper discusses an architecture to allow for policy definition and call control by policies and describes the operation of a system based on this architecture. One aspect is policy definition, the APPEL policy description language that serves this purpose. An important aspect of the architecture is integral feature interaction handling. It is in this last aspect that we foresee a role for agents, and hope that this paper will stimulate some collaboration between the two mostly distinct research areas of feature interaction and agent technologies.

1 Motivation

Telecommunication has gained a central role in our daily life, be it private or within the enterprise. We have come a long way from just connecting two users and allowing them to have a verbal conversation. Over recent years we encountered a trend towards merging different communication technologies such as video conferencing, email, Voice over IP as well as new technologies like home automation and device control. This is combined with a move to greater mobility, e.g. wireless communications, mobile telephony, *ad hoc* networking, as well as traditional landlines. *Features* such as conference calling and voice mail have been added to help users deal with situations beyond simple telephony.

Currently such features only support communication, that is they allow the user to simplify and more closely integrate telecommunication in their activities. In the future, features will make use of the merged technologies on any device. We believe that features then will enable communications by allowing the user to achieve particular goals.

From this we conclude that the end-user must have a central place in communication systems. An obvious extension to this is that any feature must be highly customizable by end-users, as it is only the individual who is aware of

* The material published in this paper has been developed in the ACCENT project during the authors previous employment at the University of Stirling.

M.D. Ryan et al. (Eds.): Objects, Agents, and Features, LNCS 2975, pp. 189–208, 2004.

his requirements. It might even be desirable to allow the end-user to define his own features. However, end-users are usually not computer experts, so any customization or feature development must be simple and intuitive.

Policies have been used successfully in system administration and access control as a way for users to express detailed preferences. We believe that policies written by – and thus under the control of – individual users provide the functionality required.

We discuss the advantages of policies over features, and define an architecture in which policies can be used to control calls. We will find that policies do not remove the feature interaction problem, and hence conflict detection and resolution mechanisms will form an essential part of the proposed system. Resolution of conflicts at runtime requires flexible mechanisms, such as those that *agent* technologies can provide. The system is user-oriented and addresses the fact that future call control needs to enable individuals to achieve their goals.

We provide the background to this work in section 2, in particular introducing the key concepts: features, policies, feature interaction and policy conflict. In section 3 we propose an architecture and discuss its components. Sections 4 and 5 are used to define the operation of a system built on the architecture. We then return to discuss the link between policies and agents in section 6. Finally we summarise the approach and indicate further work in section 7.

2 Background

2.1 Policies and Features

Policies are defined as *information which can be used to modify the behaviour of a system* [16]. We interpret *modify* in this definition, to include specification of default behaviour, restriction of the system and enhancements of the functionality. Considerable interest in policies has developed in the context of multimedia and distributed systems. Policies have also been applied to express the enterprise viewpoint [22] of ODP (Open Distributed Processing) and in agent-based systems [4].

In telecommunication systems, customized functionality for the user is traditionally achieved by providing features, i.e. capabilities that have been offered on top of a basic service. Features are imperative by nature, in contrast to policies which are descriptive. Features are supplied by the network provider and thus do not offer completely customized functionality. Consider a call forwarding feature: the user choices are whether the feature is available or not, and which telephone number the call gets forwarded to[1]. There is no possibility for the user to forward only some calls or to treat certain calls differently, e.g. forward private calls to the mobile and others to a voicemail facility.

However, recent technological advances mean that users are connected nearly anywhere and nearly all the time. In this context, users should have much more

[1] Note, however, that large organisations can have "call plans" that allow for routing based on time and location.

control over their communications. Users should be able to define their availability, that is when they are available to receive communications and which ones. However, a usable mechanism for this must go much further than static rules; it must take the current context of the user into account. After all, the goal of new features should be to provide communications that are least intrusive to the user, but at the same time trying to satisfy all parties involved.

It is exactly the flexibility, adaptability and end-user definition of policies that makes them an ideal candidate technology for features of the future.

2.2 Policies and Feature Interaction

One might hope that policies would remove the feature interaction problem, simply by being higher-level. However, this would be far too optimistic, and in fact interaction problems are a harsh reality even at the more abstract level. The policy community has recognised that there are issues, which they refer to as *policy conflict*, but has not considered any practical solutions (the best effort to classify the problem being [16]). Most people in the policy community appear to think that this is a minor problem and will simply vanish, an opinion shared by many others outside the feature interaction community. We will use the terms policy conflict and feature interaction interchangeably, as essentially the two refer to the same problem but in two different domains.

The difference between policy conflict and feature interaction is twofold. The future will see many more policies than we have seen features. This is due to policies being defined by end-users. In addition to the increase in the number of policies, users are not aware of the feature interaction problem, or at least are generally not expert enough to anticipate problems. It should be obvious that this only increases the size of the feature interaction problem. It requires more workable, automatic solutions to detecting and resolving interactions. On the other hand, policies can contain preferences and the context can also provide priorities which can be used to resolve conflicts. The new communication architectures provide richer underlying network technologies that more readily allow information exchange between parties. Thus one could argue that the feature interaction problem is actually easier to deal with in new systems.

2.3 Feature Interaction and Agents

One of the strong aspects of agents is their ability to cope collaboratively with unexpected events in the environment due to some of their basic criteria: autonomy, pro-activeness, reactivity, collaboration and negotiation. To achieve this agents use standard AI techniques derived from knowledge engineering and machine learning, but add distributed coordination and negotiation facilities. An introduction to intelligent agents is provided in [27]. The occurrence of feature interaction is an unexpected event in a distributed environment, so agent technology might provide solutions here.

There is some work in the feature interaction community discussing the role of agents. Features and resources are represented by agents able to communicate

with each other to negotiate their goals; successful negotiation means that an interaction has been resolved. The *a priori* information required is hidden in the concept of successful negotiation – it must be known when goals interfere.

In an early paper [25], Velthuijsen evaluated a number of distributed artificial intelligence techniques (DAI) to help solve the feature interaction problem. Several approaches have since been developed using the techniques. For example, Griffeth and Velthuijsen use negotiating agents to detect and resolve interactions in [12]. A resolution is a goal acceptable by all parties, and is achieved by exchanging proposals and counter-proposals amongst the agents. Different methods for negotiation have been envisioned: direct (agents negotiate directly without a negotiator), indirect (a dedicated negotiator controls the negotiation and can propose solutions based on past experience) and arbitrated (an arbitrator takes the scripts of the agents and has sole responsibility to find a solution). Griffeth and Velthujsen concentrate on indirect negotiation. The approach has been implemented in the experimental platform "Touring Machine" [3], although no conclusive report about the success is provided.

Rather than using direct negotiation, Buhr et al. [6] make use of a blackboard. Features are represented by agents which exchange information by writing to a public data space. Other agents can change the information written to the blackboard and a common goal can be negotiated. Amer et al. [1] also use the blackboard technique, but extend their agents to make use of fuzzy policies. Agents set truth-values (0 to 100) to express the desirability of certain goals. These values are then adapted as the call progresses, depending on the values of other agents. In the case of a conflict, an event with the highest truth-value is executed.

The suggested approaches are promising. But the existing telecommunication network architectures do not provide feature-to-feature communication. The approaches require a significantly different network architecture. We will discuss such an architecture in the policy context and consider the link between agents and policies later in section 6.

3 The Policy Architecture

In [21] we proposed a three layer architecture consisting of (1) the communications layer, (2) the policy layer and (3) the user interface layer. We will discuss the relevant details of the architecture again in this paper. A three-tier architecture is used in completely different ways for other applications. However, a three-tier policy architecture emerges naturally.

This architecture provides a number of key aspects, the details of which we discuss throughout the paper. With reference to Fig. 1:

- The Policy Servers can communicate with each other to negotiate goals or solutions to detected problems. This is not possible in existing communication architectures. It is this element that makes the use of agent technologies possible.

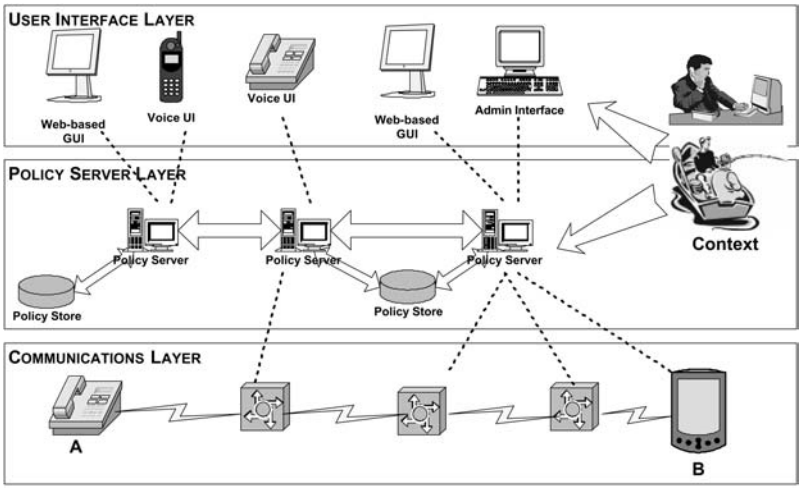


Fig. 1. Overview of the Proposed Architecture

- The User Interface Layer provides end-users with a mechanism to define functionality using a policy definition language. Such a mechanism does not exist in traditional telecommunication architectures, but recent architectures such as SIP (the Session Initiation Protocol) allow for caller preferences. A similar layer could be used in agent architectures to define the agent's goals.
- The User Interface Layer and the Policy Servers make use of context information (e.g. the user is in the office) to provide new capabilities. To our knowledge this has not been done before in the communications domain, but is somewhat natural to agent technologies where the environment influences the behaviour.
- The Policy Server layer is independent of the underlying call architecture. That means that we can work across several communication architectures if required. Advances at the communications layer will not adversely impact on the higher layers.
- All intra-layer signalling can be achieved by methods based on protocols such as SOAP (Simple Object Access Protocol).

The architecture makes the underlying communication network transparent to the higher layers. It enables end-user configurability and provides communication between policy servers which can be used for interaction handling. We will briefly discuss next the details of each layer.

3.1 The Communications Layer

The communications layer represents the call architecture. Details are not relevant for this paper. However, we impose two crucial requirements on the communications layer. (1) The policy servers must be provided with any message

that arrives at a switching point; routing is suspended until the policy server has dealt with the message. (2) A mapping of low-level messages from the communications layer into more abstract policy terminology must be defined (usually developed by a domain expert, and in its simplest form is a dictionary).

In this paper we assume that both requirements are fulfilled. Our investigations have shown that this is a realistic assumption.

3.2 The Policy Server Layer

The policy server contains a number of policy servers that interact with the underlying call architecture. It also contains a number of policy stores, that is database or tuple space servers where policies can be stored and retrieved by the policy servers as required. We assume that several policy servers might share a policy store, and also that each policy server might control more than one switching point or apply to more than one end device in the communications layer. One user's policies are stored in a single policy store. For reliability replication could be considered (however this is outside the scope of the current work).

The policy servers interact with the user interfaces in the policy creation process as will be discussed in section 4. They also interact with the communications layer where policies are enforced; this will be discussed in detail in section 5.

A further role of the policy server is to detect and resolve (or suggest resolutions of) conflict among policies. To enable richer resolution mechanisms, communication between policy servers for information exchange is permitted. Further, the policy servers have access to up-to-date information about the user's context details which are used to influence call functionality.

3.3 The User Interface Layer

The user interface layer allows users to create new policies and deploy these in the network. A number of interfaces can be expected here: graphical or text-based interfaces, voice-controlled mechanisms, or administrative interfaces, each providing functionality targeted at certain types of users and devices.

The normal user will use a text-based interface for most functions. However mobile users might not have the capability to use such technology, so voice-controlled interfaces are more appropriate. These also suit users that simply want to activate or deactivate policies. A voice interface is essential for disabled or partially sighted users. Both text and voice interfaces should guide the user in an intuitive way, preferably in natural language or in a graphical fashion. We discuss a text-based interface in more detail in section 4. We also envision libraries of policy elements that users can simply adapt to their requirements and combine to obtain the functionality required, in a similar way that for example clip-art libraries are common today.

The administrative interfaces are system-oriented and exist mainly for system administrators to manage more complex functionality.

The policy definition environment provides access to contextual variables which are instantiated at runtime by the policy servers with actual context

information. For example, "secretary" in a forwarding policy could be filled in from a company organisation chart in conjunction with holiday and absence information. Also, the location of a user could be determined from an active badge system.

4 Defining Policies

Policies should provide the end-user with capabilities to get the most out of their communication systems. End-users usually use their communication devices in a social or commercial context that imposes further policies. For example an employee is often subject to company policies. Hence policies will be defined at different domain levels (users, groups, companies, social clubs, customer premises, etc.) by differently skilled users. Any policy definition process needs to take this into account.

4.1 A Policy Description Language

In previous work [20] we have introduced initial ideas for a policy description language (PDL) that allows us to express call control policies. Here we present more details on APPEL (the Accent Project Policy Environment/Language)[2]. We have analysed a set of more than 100 policies before defining a language that is able to express these. Further, any traditional feature can also be expressed.

The syntax of APPEL is defined by an XML grammar which we present using a graphical notation. We will only provide an informal insight into the semantics where appropriate.

A policy document (`policydoc`, see Fig. 2) forms the highest level in which one or more policies are encapsulated.

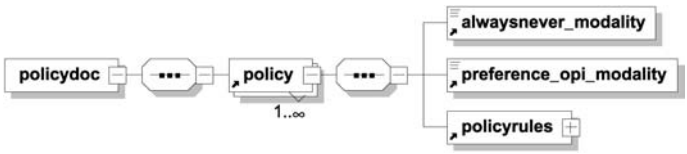


Fig. 2. The Policy Document Element and its immediate Children

A `policy` is the object of highest interest in the policy description language. A policy has a number of attributes, whereby `owner` highlights the person that the policy belongs to and `appliesTo` highlights the person/group that the policy applies to. Here one could argue that the two are mostly the same. However, in an enterprise context this might not be so: a policy might have been created (and be owned by) a system administrator but applies to one or more people in

[2] appel – from French: a call

the enterprise. Further each policy can be enabled or not, determining whether the system should apply the policy.

A policy contains a `policyrules` element and might contain two kinds of modalities. Note that the user interface will typically also provide for temporal modalities – however these are encoded as conditions. A policy might specify an always/never modality. If always is specified it is assumed that the policy actions should be applied. If never is specified the user intends that the actions do not occur. Policies might also have a preference or obligation modality. These modalities will be used when policy conflict is detected and must be resolved. We have currently not specified a weight for the preference modalities.

A policy contains one or more policy rules (`policyrules`), where the policy rules are concatenated using one of four operators: guarded choice, unguarded choice, parallel application or sequence. All these operators are binary; larger chains can be achieved by appropriate nesting of policy rules.

When evaluating policies, it is relevant to be able to determine if a policy applies and if so, what actions should be taken. This depends fundamentally on the meaning of the combination operators. We discuss this next:

Guarded choice. When two policy rules are joined by the guarded choice operator, the execution engine will first evaluate the guard. If the guard evaluates to "true", the first of the two rules will be applied, otherwise the second. Clearly once the guard has been evaluated it is necessary to decide whether the individual rule is applicable and whether there is no conflict that prohibits the enforcement. The guard could be emulated by suitable conditions on the level of a policyrule, however there are uses for having guards at the higher level.

Unguarded choice. Unguarded choice provides more flexibility, as both parts will be tested for applicability. If only one of the two policy rules is applicable, this will be chosen. If both are applicable, the system can choose one at random (the current implementation will select the first).

Sequential composition. With sequentially composed policy rules, the rules are enforced in the specified order. That is we traverse the structure, determining whether the first rule is applicable. If so, we apply the first rule, and then move on to check the second rule. Note that the second rule will only be checked if the first rule is applicable.

Parallel composition. Parallel composition of two rules allows for a user to express an indifference w.r.t. the order of two rules. Both rules are enforced, but the order in which this is done is not important. An example would be a rule that leads to the attempt to add a video channel and the logging of the fact that this is attempted: we would allow the system to resolve the order of the two actions.

A policy rule (see Fig. 3) defines the triggers and conditions that need to hold for the action to be applied. We allow for policy rules that do not have trigger events (so-called goals) and also for rules that do not specify conditions. However, an action always needs to be specified.

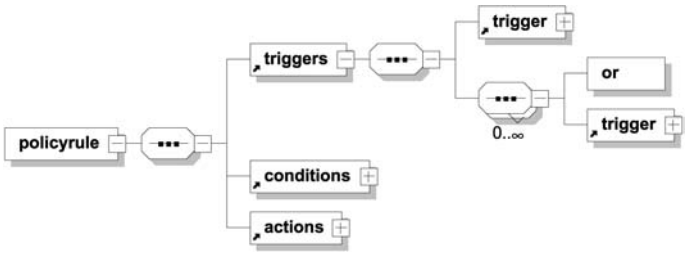


Fig. 3. The PolicyRule Element and its immediate Children

A policy might be activated by one or more triggers. The semantics here is that if any of the stated triggers occurs during enforcement, the policy is considered to have been triggered – whether the action should then be applied depends on the conditions provided. Goals are always considered to be triggered.

The enforcement of a policy usually means that the actions stated within the policy are applied to the underlying system. A user can specify a single action or a number of actions. If a number of actions is specified it is important to determine their application order. Four operators are provided for this purpose; we discuss their respective meaning next.

And specifies that the policy should lead to the execution of both actions. However, the order is not specified. This means that in general we will execute `action1` and `action2` in any order. The current implementation will execute the first action first, but this is only a design decision and one could execute the actions in parallel.

Or specifies that either one of the actions should be taken. The current implementation will assume that `action1` will be taken. Note that "or" is important when a conflict is detected: it provides the system with an alternative action to be taken which might not be conflicting.

Andthen is a stronger version of "and", as here the order is prescribed in that `action1` must precede `action2` in any execution.

Orelse is the "or" operator with a prescribed order. This specifies that a user requires `action1` to be tried first.

We have discussed the actions and triggers elements of a policy rule; now we consider conditions. A condition might be a simple condition or a more complex combination of `conditions` (see also Fig. 4). In the latter case, conditions can be combined with the usual boolean "and", "or" and "not" operators. These operators have their usual meaning.

A `condition` is a simple test where a parameter is compared to a value using a number of operators. The operators are predefined in the grammar, but might have subtly different meaning depending on the arguments they are applied to. For example `in` applied to time might mean that the actual time falls within an interval, but `in` applied to a domain object might mean that the entity is in the domain. Conditions in policies refer to concepts from the context of the

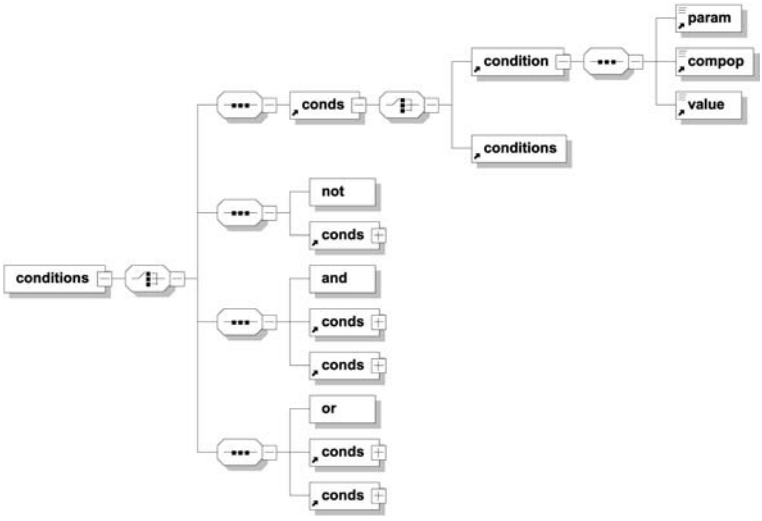


Fig. 4. The Conditions Element and its immediate Children

user/call. These concepts are encapsulated in parameters. It is possible to add further parameters.

Note that while the language has been developed in the context of call control systems it is thought to be more generally applicable. The only aspects of the language that are domain-specific are the available parameters, triggers and actions. However, these are clearly separated and thus can be readily adapted to other domains.

4.2 User Interfaces

The presented language is quite expressive. However, we do not expect the end user to define policies directly in the policy description language (i.e. raw XML. This would be unrealistic, as we expect the user to be less knowledgable and certainly not to be a computer expert. In order to make the policy language usable, we have developed a simple wizard that aids the user in creating and handling policies (currently a prototype). Also, this wizard is specific to policies in the call control context, but can be adapted to other domains. The interface is web-based, so users can access it from anywhere. The users are authenticated, and depending on their credentials will be allowed more or less possibilities in the interface. In general the interface provides functionality to create new policies, edit or delete existing ones as well as enabling and disabling policies that are currently deployed.

Once the user submits a change request the system gathers the relevant information. For creation and editing of policies, the XML representation of the policy is generated. The policy identifier, and if required the XML version, are submitted to the policy server. We consider the details in section 4.3.

It is intended that further interfaces will be added. In particular we will investigate VoiceXML based interfaces to allow users to handle policies on devices such as mobile phones where speech is the most appropriate input medium. At the lowest level, administrators might encode policies directly in XML using off-the-shelf tools.

4.3 Upload of Policies

A policy server provides an interface to receive changes to policies (with appropriate authorisation). The current implementation receives simple text messages on this interface, although we consider using SOAP [26] for future enhancements. The user interface layer uses this interface to submit the gathered information and to receive any feedback on success or failure of the operation.

In the absence of the feature interaction problem, all that would be required now is to insert the new policy into the policy store or to update the existing policy. This is exactly what happens if no conflict is detected. However as we have indicated, conflicts are rather likely. Hence, before inserting policies into the store a consistency check is performed on the set of policies applicable to the user. If a conflict is detected the change is not committed to the policy store, but rather an informative response is sent to the user. Such a message will state that there was a problem and suggest possible solutions, if they are available.

4.4 Interactions

It would be desirable if the policy architecture would not give rise to conflicts. However, this holy grail has not been achieved in many years of feature interaction research. It appears to be possible only at the cost of reducing the expressiveness of the policy language to trivial levels. We have taken a more realistic stance by introducing a guided design process that automatically checks policies for the presence of conflict and presents any detected problems to the user, together with suggested resolutions.

When the user has attempted to upload a new policy this is checked against other policies from the same user, but also against policies that the user might be subject to (e.g. due to her role in the enterprise). We can however only check for static interactions, i.e. those that are inherent to the policies independent of the contextual data. This suggests the use of offline detection methods and filtering techniques, which have been developed by the feature interaction community.

Any inconsistencies detected need to be reported to the user. The resolution mechanism is typical of offline methods – a redesign of the policy base. We consider the following methods most appropriate for the policy context, although an implementation is required to determine which is most suitable.

Anise [24] applies pre-defined or user-defined operators to build progressively more complex features. This neatly maps onto the philosophy of the policy definition language. The composition mechanism is able to detect certain forms of interaction arising from overlapping triggers. Zave and Jackson's [28] Pipe and

Filter approach detects conflict of features that occur in specific, pre-defined order. An interesting approach is presented by Dahl and Najm [9] which allows for ready identification of interactions as occurrence of common gates in two processes. Thomas [23] follows the more common approach of guarded choices. The occurrence of non-determinism highlights the potential for interactions. Available tools allow the automatic detection of non-determinism, and these tools can be incorporated into the policy server. Many of the off-line techniques require analysis. This is done by either checking the models against specified properties (e.g. feature and system axioms) or by finding general properties such as deadlock, livelock or non-determinism (or a combination of the three). Again automatic tools for detecting these problems can be incorporated as part of the policy server. Note that these methods are applied at policy definition time, so execution times are less of an issue (as long as they stay within reason).

In fact we are not restricted to static interactions: we can also detect the potential for conflict. That is, we can filter cases where an interaction might occur, depending on the particular instantiation of contextual data. A suitable approach might be derived from the work of Kolberg *et al.* [14] Here the user has a choice of redesigning the policy or simply to accept that conflict might occur and let the enforcement process handle it when it occurs. In the latter case some additional information might be included in the policies to indicate the potential conflicts to guide any online resolution technique. The question, as to what information is required remains unanswered in our work so far.

One might suggest that the consistency check be performed at the user's device rather than at the policy server – a valid point which has been considered. Performing the check in the policy server has the major advantage that all details of the user's policies, as well as those from the same domain, are accessible and can be considered without transferring all policies to the user's device. Furthermore the user side of the implementation is kept light-weight, which is important for less powerful end-devices such as mobile phones or PDAs.

4.5 A Worked Example

To illustrate what we said in this section, consider the following example with a typical user, say Alice.

Alice already has a policy that asks to "forward calls to my voicemail in evenings". However, she is expecting an important call and sets up a new policy that specifies "forward calls to my mobile after 16:00".

To set up the policy, she will chose an appropriate policy type *forward calls to target*, which will allow her to set *target* to "mobile", simply by selecting values from lists. She then can add an extra condition, again by choosing a prototype *when it is later than time* and refining time to "16:00".

On receiving an upload command, the policy server checks the policies for consistency. Under the intended semantics, Alice's new policy conflicts with her existing policy: 'in the evening' both apply and they specify different forward targets. A possible solution that might be suggested is that Alice disables her usual evening policy temporarily to allow for the exceptional circumstances.

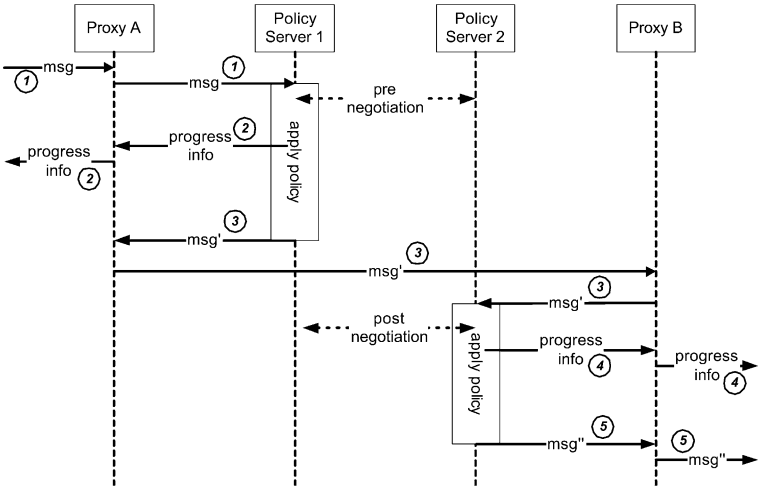


Fig. 5. Policy Enforcement Process

5 The Call Process

In the previous section we have discussed how users can define their policies and upload them to the policy server. In this section we discuss how policies are enforced to achieve the goals they describe.

5.1 Applying Policies to a Call

Let us consider a typical call setup. When attempting to set up a call from A to B, A's end device will generate a message that is sent to A's switching point (or proxy). From there the message gets forwarded through a number of further switching points until it reaches B's end device.

If we assume that policy servers are associated with switching points, then at every switching point that the message reaches it is sent to a policy server. Routing is suspended until the policy server allows continuation, and only then is continued with respect to the response from the policy server. In SIP, we can intercept, modify, fork and block messages by using SIP CGI [15]. It is easy to extract the complete message and pass it to the policy server for processing.

Once the policy server has obtained a message it needs to process it. The general process is to retrieve all applicable policies and to change the message as required. If there are no applicable policies, the message is simply returned to the proxy unchanged for onward routing. More interestingly, if one or more policies apply the required changes can be made to the original message. Alternatively new or additional messages can be created. The example in Fig. 5 shows "progress info" messages that are generated by the policy server. Also, if a policy requires forking of a call (e.g. "always include my boss in calls to lawyers") the respective messages to set-up the extra call leg need to be created.

Any messages returned to the proxy server are then routed onwards to the next proxy server, where the same process of forwarding the message to a policy server and application of policies is performed. Figure 5 shows an example of this process for a call between two parties where policies are enforced in one direction. We can also have scenarios where the policy at the remote end affects the originating end. Note that the figure shows two message exchanges between the policy servers, labelled pre- and post-negotiation. These allow for resolution of conflicts; we return to them later in section 5.3.

The process of applying a policy is somewhat complex. It requires the received message to be analysed, then the retrieval of any applicable policies as well as the retrieval of any required context information. Conflicts of policies must be determined, and new messages must be generated and sent.

It is impractical to require the user to always provide the relevant information when establishing a call, but this is not necessary. Most of the required information can be inferred from the context. For example, roles may be defined in a company organisation chart, the location can be established from the user's diary or mobile home location register. Or better, "mobile devices have the promise to provide context-sensing capabilities. They will know their location – often a critical indicator to the user's tasks." [11]. From the location, further facts can be derived, depending on the quality of the location information, GPS does not allow fine grained positioning, but smart badges could precisely locate users inside buildings. Hence, from you being in your boss's office, it can be derived that you are probably in an important meeting and do not wish to be disturbed. When having lunch with friends or colleagues, a lecturer might not wish to be disturbed by students. Note that we are not concerned with privacy issues at this moment.

The analysis of the messages must extract the relevant information, such as sender and recipient, as well as any other information that might occur in policies. Here it is important to point out that this involves a translation of communications layer messages into policy events. Clearly, the amount of call status information that can be used is dependent on what the underlying communications layer supports. For example, in the IN the topic of a call is usually not identifiable and hence cannot be used. Messages that are issued from the policy server to the communications layer require a reverse translation, that is policy events need to be converted into communications layer messages.

A number of policies can be applicable in any scenario. All goals and those policies relating to the trigger event can be activated by the call, provided that the sender or target of the message applies to the user of the policy.

In a next step further conditions must be evaluated. Usually this requires the context variables in the conditions to be instantiated with current values extracted from context information. Those where the conditions are not satisfied can be discarded from the set of applicable policies.

The remaining set of policies needs to be analysed for conflicts, and these must be resolved before the response messages can be generated. We consider an example before reconsidering interactions.

5.2 A Worked Example

We return to Alice for a further example. We now also have a company competition.com with three employees Paul, John and Janine.

Alice regularly speaks to Paul but also knows John. She prefers to speak to people she knows, so she has a policy that states "I prefer to speak to John if Paul is busy". But Paul also has a policy to handle redirection when he is busy: "I expect my calls to be redirected to Janine when I am busy".

Alice rings Paul while Paul is busy, this causes a conflict as Alice prefers to be forwarded to John rather than Janine. This conflict is somewhat new with respect to feature interactions, it is caused by a caller preference. This was not possible with traditional features.

For a resolution we can see several options here. One could always give precedence to caller expectations (after all, they are usually the paying party) – in which case Alice gets her wish. We also have preferences in the policies, and here Alice has expressed "prefer", whereas Paul has expressed a rather stronger concept of "expect" – thus, Paul will be granted his goal.

Another option is to start a negotiation process. Details of how to do this are not clear at this point. It is here that we hope for agent technologies to provide solutions; we return to this when discussing interactions in the next section.

5.3 More Interactions

In the example we have seen an interaction that only occurred while a call was actually taking place. Any detection mechanism incorporated in the enforcement part of the policy server needs to be able to detect and resolve such conflicts. The introduction of policy support must also not create unreasonable delays in call setups. However, if users are aware that complex steps are needed to resolve sophisticated policies, they should learn to live with the delays – and probably are happy to do so when the outcome is productive for them (hence the intermediate information messages are produced by the policy application process).

On-line and hybrid feature interaction approaches could provide possible solutions. A special category of on-line techniques, so called hybrid techniques, provides information gathered by an off-line phase to improve the on-line technique. This is of particular relevance when the available information at run-time is too limited to resolve detected interactions. However, we believe that in a policy context the available information is sufficient to resolve conflicts. The underlying architecture provides protocols that are rich enough to facilitate exchange of a wide range of information. There are essentially two classes of run-time approaches. One is based around the idea of negotiating agents, the other around a feature manager. Examples of the former are presented by Velthuijsen [25] and Buhr et al. [6]. Examples of the latter are presented by Cain [7], Marples and Magill [17], and Reiff-Marganiec [19].

In the feature manager approach a new entity is included in the call path, a so-called feature manager that can intercept events occurring in the system (such as

messages sent, access to resources, etc.). Based on this information, the detection and resolution of conflict is possible. In [7], the feature manager knows specific undesired behaviour patterns and potential resolutions. The feature manager in [17] detects interactions by recognising that different features are activated and wish to control the call. The resolution mechanism for this approach [19] is based on general rules describing desired and undesired behaviour.

Feature manager approaches lend themselves to the policy architecture, as their main requirement is that the feature manager is located in the call path. This is naturally the case with policy servers. However, feature manager approaches so far have suffered from a lack of information to resolve interactions (though some progress has been reported in [19]).

Negotiation approaches have been considered in section 2. Their handicap in current architectures is the sophisticated exchange of information required which is not supported. However, our architecture provides for this by allowing communication channels between policy servers.

Two forms of negotiation are practical in the policy architecture: pre- and post-negotiation. In the former case, the policy server contacts the policy server at the remote end and negotiates call setup details to explore a mutually acceptable call setup. The communications layer is then instructed to set up the call accordingly. In post-negotiation, the policies are applied while the call is being set up, thus potentially leading to unrecoverable problems.

We hope that the agent community can provide solutions for negotiation in this context – or help with their development. Clearly, negotiation requires the exchange of information. It is here that we have a number of immediate points: it must be researched what information needs to be exchanged for negotiations to be successful. Also, as we are dealing with personal policies it might be undesirable to make information available to the other party. Clearly issues of privacy (which could be seen as "appropriate sharing" rather than hiding of information) need to be considered in any solution. Also, any approach developed will need to be part of the production system and hence must be efficient and scalable.

6 Policies and Agents

This short section simply serves the purpose of bringing the discussion from the background full circle. We have earlier described how policies and features are related, as we discussed the relation between feature interaction and agents and have also (throughout the paper) highlighted the similarities between feature interaction and policy conflict. Here we will summarise the relation between policies and agents.

We can consider the relation between policies and agents in two ways: *how can agents help with policy conflict?* and *can policies be useful for agents?* We will not attempt to provide definite answers to either question, but rather give our general view – in this sense the following is rather speculative.

In section 5.3 we have discussed the role that we foresee for agents in the policy architecture. Our hope is that agents will provide the solution to the

negotiation required at runtime when a policy conflict is detected, we called this post-negotiation. We have also indicated that agents might attempt to prevent conflict, by determining outcomes before the actual actions are committed. We referred to this as pre-negotiation.

In the former case it is required of the agents to exchange the relevant details to come to a solution. However, note that we are not dealing with just two agents: there might be many more with each representing an entity that is interested in the call. In particular there are the end-users, but there might also be companies that impose rules and network providers that have to ensure that the communication network remains stable. One could envision decentralised negotiation or, if this is more suitable, centralised negotiation via some trusted "black board" or other entity. However, we can see that negotiation is relatively independent from the underlying call architecture as all that must be known about the network is the entities that have an interest in the negotiation.

The latter case does make matters somewhat more complex: a closer integration between the call architecture and the agents is required. In addition to what was said above the entities that have an interest in the communication must be found. While a remote end of the call (usually) knows the route the call took to reach it, the originating end cannot know the route. This is largely because the obvious route might not be appropriate in the context of policies that change the route (for example a forwarding policy).

However, we can also see that a policy framework and language (like APPEL) might be useful for users to describe their goals for agents. In general agents are acting on behalf of a user to achieve goals that the user wishes to achieve. However, as far as the author understands, these goals are often encoded in the agent in a way that is similar to what features provide for telecommunications. For example, a search-bot has the goal of finding information regarding a certain query, and it is the query that can be specialised by the user. We think that user defined policies could provide a means of making agents more generic and dynamically reconfigurable. In particular they could be controlled by high-level goals set by the user. These goals will then automatically provide a framework for negotiation, as all policies together specify what is and what is not acceptable for the user. This requires for the policy language to be connected to agent ontologies in a generic way.

We believe that both aspects require some further investigation, which might best be conducted by teams composed of members of both the agents and the feature interaction communities.

7 Conclusion and Further Work

7.1 Evaluation

We have considered how policies can be used in the context of call control, especially how they can be seen as the next generation of features. The policy architecture allows calls to be controlled by policies. Each policy might make use of the context of a user. This allows context-oriented call routing, but goes far

beyond routing by allowing for availability and presence of users to be expressed. In this way we can achieve truly non-intrusive communications that enable users to achieve their goals.

Policies can be easily defined and changed. We have introduced the APPEL policy description language. More importantly, changes and definition of new policies can be performed by the end-user via a number of interfaces. Web and voice interfaces have been discussed.

We have suggested some off-line techniques from feature interaction for policies to be checked for consistency when they are designed. They can be applied to new policies as well as to existing ones where detailed information is available (e.g. within the same domain). However, calls will eventually cross domains and then it is not possible to determine the policies that might conflict due to the sheer number of policies and users; the actual connections between users cannot be predicted. This requires that we make use of on-line techniques to resolve any such issues. To detect conflicts automatically, both off-line and on-line methods require an understanding of the conditions and actions to determine when policies are invoked simultaneously and when their actions lead to inconsistencies.

7.2 Future Work

A prototype environment to create and enforce policies has been developed on top of a SIP architecture. Thus the proposed architecture has been implemented.

The methods for detecting and resolving policy conflict identified in this paper need to be implemented in the prototype such that empirical data on their suitability can be gathered. That is, policy servers need to be equipped with mechanisms that can perform static consistency checking of uploaded policies. They also require mechanisms to detect and resolve conflicts at run-time. It is here that we envision the use of agent technologies. We have discussed some of the requirements and open questions in section 5.3.

In the future we would like to test the top layers on top of other systems. For example, in the telecommunication domain we are considering H.323, but we would also like to move the concept outside the telecommunication domain and have planned an investigation in the context of web services. Further development of additional user interfaces should strengthen the prototype. Another research area is the automatic gathering of context details, which is interesting in itself but beyond the scope of our work.

Other ideas for further work, include the linking of agent ontologies to the policy language in a generic way (as suggested in section 6). Also, the issue of privacy has not been addressed in this paper. For the acceptance of the system, secure storage and controlled access to user policies and profiles is highly important. Also, legal reasons prescribe for any system containing such private information to have a clear privacy “policy”.

Acknowledgements

This work has been supported by EPSRC (Engineering and Physical Sciences Research Council) under grant GR/R31263 and Mitel Networks Corporation.

The work has been conducted as part of the ACCENT project during the authors previous employment at the University of Stirling. I thank all people who contributed to the discussion of policies in this context. Particular thanks are due to colleagues at Mitel Networks Corporation and at Stirling University, in particular to Ken Turner who strongly influenced this work. Thanks are also due to Jennifer Paterson for implementing the policy wizard.

References

1. M. Amer, A. Karmouch, T. Gray, and S. Mankovskii. Feature interaction resolution using fuzzy policies. In *[8]*, pages 94–112, 2000.
2. D. Amyot and L. Logrippo, editors. *Feature Interactions in Telecommunications and Software Systems VII*. IOS Press (Amsterdam), 2003.
3. M. Arango, L. Bahler, P. Bates, M. Cochinwala, D. Cohrs, R. Fish, G. Gopal, N. Griffeth, G. E. Herman, T. Hickey, K. C. Lee, W. E. Leland, C. Lowery, V. Mak, J. Patterson, L. Ruston, M. Segal, R. C. Sekar, M. P. Vecchi, A. Weinrib, and S. Y. Wuu. The Touring Machine System. *Communications of the ACM*, 36(1):68–77, 1993.
4. M. Barbuceanu, T. Gray, and S. Mankovski. How to make your agents fulfil their obligations. *Proceedings of the 3rd International Conference on the Practical Applications of Agents and Multi-Agent Systems (PAAM-98)*, 1998.
5. L. G. Bouma and H. Velthuijsen, editors. *Feature Interactions in Telecommunications Systems*. IOS Press (Amsterdam), 1994.
6. R. J. A. Buhr, D. Amyot, M. Elammari, D. Quesnel, T. Gray, and S. Mankovski. Feature-interaction visualization and resolution in an agent environment. In *[13]*, pages 135–149, 1998.
7. M. Cain. Managing run-time interactions between call processing features. *IEEE Communications*, pages 44–50, February 1992.
8. M. Calder and E. Magill, editors. *Feature Interactions in Telecommunications and Software Systems VI*. IOS Press (Amsterdam), 2000.
9. O. C. Dahl and E. Najm. Specification and detection of IN service interference using LOTOS. *Proc. Formal Description Techniques VI*, pages 53–70, 1994.
10. P. Dini, R. Boutaba, and L. Logrippo, editors. *Feature Interactions in Telecommunication Networks IV*. IOS Press (Amsterdam), 1997.
11. A. Fano and A. Gersham. The future of business services in the age of ubiquitous computing. *Communications of the ACM*, 45(12):83–87, 2002.
12. N. D. Griffeth and H. Velthuijsen. The negotiating agents approach to runtime feature interaction resolution. In *[5]*, pages 217–236, 1994.
13. K. Kimbler and L. G. Bouma, editors. *Feature Interactions in Telecommunications and Software Systems V*. IOS Press (Amsterdam), 1998.
14. M. Kolberg and E. H. Magill. A pragmatic approach to service interaction filtering between call control services. *Computer Networks: International Journal of Computer and Telecommunications Networking*, 38(5):591–602, 2002.
15. J. Lennox, J. Rosenberg, and H. Schulzrinne. Common gateway interface for SIP. *Request for Comments 3050*, Jan 2001.
16. E. C. Lupu and M. Sloman. Conflicts in policy-based distributed systems management. *IEEE Trans. on Software Engineering*, 25(6):852–869, 1999.
17. D. Marples and E. H. Magill. The use of rollback to prevent incorrect operation of features in intelligent network based systems. In *[13]*, pages 115–134, 1998.

18. D. Peled and M. Vardi, editors. *Formal Techniques for Networked and Distributed Systems – FORTE 2002, LNCS 2529*. Springer Verlag, 2002.
19. S. Reiff-Marganiec. *Runtime Resolution of Feature Interactions in Evolving Telecommunications Systems*. PhD thesis, University of Glasgow, Department of Computer Science, Glasgow (UK), May 2002.
20. S. Reiff-Marganiec and K. J. Turner. Use of logic to describe enhanced communications services. In *[18]*, pages 130–145, November 2002.
21. S. Reiff-Marganiec and K. J. Turner. A policy architecture for enhancing and controlling features. In *[2]*, pages 239–246, June 2003.
22. M. W. A. Steen and J. Derrick. Formalising ODP Enterprise Policies. In *3rd International Enterprise Distributed Object Computing Conference (EDOC '99)*, University of Mannheim, Germany, September 1999. IEEE Publishing.
23. M. Thomas. Modelling and analysing user views of telecommunications services. In *[10]*, pages 168–182, 1997.
24. K. J. Turner. Realising architectural feature descriptions using LOTOS. *Networks and Distributed Systems*, 12(2):145–187, 2000.
25. H. Velthuijsen. Distributed artificial intelligence for runtime feature interaction resolution. *Computer*, 26(8):48–55, 1993.
26. W3C. Simple Object Access Protocol (SOAP) 1.1, `http://www.w3.org/TR/SOAP/`. Visited on 09-12-2002.
27. M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2):115–152, 1995.
28. P. Zave and M. Jackson. New feature interactions in mobile and multimedia telecommunication services. In *[8]*, pages 51–66, 2000.

Agents and Coordination Artifacts for Feature Engineering

Alessandro Ricci

DEIS
Università degli Studi di Bologna a Cesena
Via Venezia, 52, 47023 Cesena (FC), Italy
mailto:aricci@deis.unibo.it

Abstract. Agents and Multi-Agent Systems (MAS) are considered a suitable paradigm for engineering complex systems. Interaction is a primary source of this complexity and coordination plays a key role for its management, providing the means for modelling and shaping the agent interaction space so as to engineer the global behaviour of the system as a whole. Feature-based systems are certainly complex systems: they are generally composed by several interrelated parts which work together to provide global functionalities. The set of functionalities is not fixed, but typically evolves with the (dynamic) integration of new features. In this paper we investigate the possibility of using agents and coordination abstractions for the engineering of feature-based systems; in particular, typical feature issues – such as feature plug-and-play and the feature-interaction problem – are framed in the agent-based engineering context.
The content of the paper is articulated as follows: First, we provide an overview of agent-oriented software engineering, in particular of the reason why agent paradigm is suitable for complex systems. Here, the importance of the coordination dimension is reminded, in particular the adoption of suitable coordination artifacts to engineer collective behaviour of MAS is discussed. Then, features and feature-based systems are briefly considered, sketching some of the main characteristics which make them complex systems. After that, a perspective for engineering feature-based systems on top of agents and coordination artifacts is provided; the main points of the discussion are exemplified using the TuCSoN MAS coordination model & infrastructure applied to a simple case study typically found in the feature literature, an email service engineering.

1 Agents and Coordination for Complex System Engineering

Agent and Multi-Agent System (MAS) paradigms are considered the right level of abstraction for modelling and engineering complex systems, characterised by organisation structures and coordination processes that are more and more articulated and dynamic [27, 29]. Agent-based control systems, inter-organisational workflow management systems, agent-based Computer Supported Cooperative

M.D. Ryan et al. (Eds.): Objects, Agents, and Features, LNCS 2975, pp. 209–226, 2004.

Work (CSCW) and team-based cooperative military contexts are prominent examples.

By complex systems we mean here systems composed by a quite large number of independent and interacting parts, typically immersed and distributed into an unpredictable/open environment (physical or non physical, such as the Internet). The openness of systems typically implies that both the set of the components of the systems, and their relationships/interaction can change dynamically, during the lifetime of the systems themselves. Distribution and concurrency are characteristics that can be found in almost any complex system.

The agent and multiagent paradigms are considered suitable since they provide abstractions (and methodologies, technologies) which naturally account for these issues. Various definitions have been given for the agent concept. An agent can be defined as an *encapsulated computer system that is situated in some environment and can act flexibly and autonomously in that environment to meet its design objectives* [29]. Among the main agent characteristics largely described in literature here we mention [10]:

- autonomy – agents have control over both their internal state and their own behaviour. This implies that an agent embeds necessarily its own control flows;
- situatedness – agents are immersed in a particular environment, over which they have a partial control and observability: they can receive inputs related to the state of their environment through sensors and they can act on the environment through effectors;
- goal orientation – agents are designed to fulfill tasks related to some specific role, in order to achieve some specific objective.

Other important characteristics are pro-activeness, that is the ability to opportunistically adopt goals and take the initiative, and reactivity, that is the ability to respond in a timely fashion to changes that occur to their environment.

It is worth remarking here some of the reasons why the agent paradigm is considered more adequate for engineering complex systems than other mainstream paradigms, such as the object-oriented and component-oriented ones [11]. Issues such as distribution of control, concurrency, openness of the environment which characterise complex systems – are directly addressed by the abstractions provided by the agent paradigm; this is generally not true in the case of object-oriented and component-oriented models. Of course, the same issues can be faced also in the context of these paradigms, but typically by adopting nonorthogonal mechanisms which cannot found a suitable specification in the computational model of the paradigm. Conversely, object-oriented and componentoriented technologies are typically used to implement agent-based technologies.

Multi-Agent Systems are ensembles of agents, acting and working independently from each other, each representing an independent locus of control of the whole system. Each agent tries to accomplish its own task(s) and, in doing so, will typically need to interact with other agents and its surrounding environment, in order to obtain information / services that it does not possess or to coordinate its activities in order to ensure that its goals can be met. However, a

multi-agent system, as a whole, is conceived to achieve more complex or wider goals than the mere sum of its component agents goals: MASs can typically be understood as *societies* of agents, where the mutual interactions between agents and with their environment leads to a useful global behaviour [3]. For this reason, another fundamental characteristic of the agent abstraction is the *social ability*, as the capability to be part of an agent society. The two main issues concerning MASs are *organisation* and *coordination*, two interrelated dimensions that are fundamental for designing, building and enacting Multi-Agent systems. Roughly speaking, organisation accounts for defining MAS structure and rules concerning this structure, often in terms of roles, and agentrole, inter-role and role-environment relationships; coordination accounts for defining the social rules and norms necessary to manage agent interaction and drive the MAS to achieve its social objectives.

In this context, MAS infrastructures are fundamental to support agent societies lifecycle, providing basic services to support both intra-agent and interagents issues, such as agent life-cycle management, agent communication and interoperability, and high-level services for MAS organisation and coordination [5].

1.1 Engineering Systems with Agents

So, agent and MAS paradigm provides models, infrastructure and technologies that can be exploited to cover all the stages of software engineering, from analysis to deployment and runtime. For the purpose, some agent-oriented methodologies – in particular for the analysis and design of systems – can be found in the state of the art: notable examples are GAIA [28], DESIRE [1], TROPOS [21], MAS-CommonKads [9] (see [8, 26] for comprehensive surveys). It must be said, however, that none of them has reached the maturity which characterises, instead, object-oriented and component-oriented approaches.

Basically, engineering a system in terms of agents involves first identifying individual and collective/social activities needed to achieve system goals, and then mapping the activities on agents and agent societies according to some organisation and coordination approach [13, 30]. In particular, *individual tasks* are associated with one specific competence in the system, related to the need to access and effect a specific portion of the environment, and carry out some specific job. Each agent in the system is assigned to one or more individual tasks, and the agent assumes full responsibility for carrying out assigned tasks. From an organisational perspective, this corresponds to assigning each agent a specific role in the organisation/society. *Social tasks* represent the global responsibilities of the agent system, typically requiring several heterogeneous competencies. The achievement of social tasks leads to the identification of global *social laws* that have to be respected and/or enforced by the society of agents, to enable the society itself to function properly and according to the global expected behaviour. From an coordination perspective, this corresponds to choose a coordination model and use the abstractions it provides to represent and enact the social laws in terms of coordination laws shaping and ruling the agent interaction space [13].

1.2 Coordination as Engineering Dimension

Coordination is then a fundamental engineering dimension of MAS. Coordination approaches can be categorised in *subjective* and *objective* [16]: in the former, all the coordination burden related to the achievement of social tasks is distributed upon agents (no mediation abstractions are used); in the latter, the coordination burden is balanced between agents and special purpose *coordination artifacts*, which are runtime coordination abstractions mediating agent (inter)action, and specifically designed to help an agent society to achieve its objectives. The behaviour of a coordination artifact is designed to embed and enact the social laws as required to achieve a social task.

The notion of coordination artifacts supporting societies in their coordination activities can be found in different disciplines, also outside computer science. According to Activity Theory, every collaborating activity occurring in any social context is always mediated by some kind of tool (artifact), in order to manage the complexity of the involved interaction and automate the coordination [24]. Coordination artifacts are meant to be provided and supported by the MAS infrastructure, according to the *coordination as a service* perspective [25]: so the infrastructure itself must provide agents not only *enabling* services – enabling for instance communication and inter-operability –, but also *governing* ones, to specify and enact norms and rules governing agent interaction. The infrastructure then can be provider also of the basic services required to use and manage coordination artifacts – in terms of creation, inspection and adaptation of their behaviour.

Summing up, engineering complex systems upon a MAS infrastructure supporting an objective coordination approach basically accounts for assigning and enacting individual tasks by means of individual agents, and using suitably engineered coordination artifacts to support social tasks and collaborating activities.

1.3 TuCSoN Coordination Infrastructure

TuCSoN (Tuple Centre Spread over the Network) is an example of MAS coordination infrastructure supporting objective coordination [19]. An overview of the TuCSoN world is depicted in Fig. 1. In TuCSoN coordination artifacts take the form of *tuple centres*, design / runtime coordination abstractions provided to agents by the infrastructure in order to enable and govern their interaction [15]. More precisely, tuple centres are *programmable* tuple spaces [15], a sort of reactive logic based blackboards; agents interact by writing, reading, and consuming *logic tuples* – ordered collections of heterogeneous information chunks – to/from tuple centres via simple communication operations (*out*, *rd*, *in*) which access tuples associatively (see Fig. 2). While the behaviour of a tuple space in response to communication events is fixed and pre-defined by the model, the behaviour of a tuple centre can be tailored to the application needs by defining a suitable set of *specification tuples*, which define how a tuple centre should react to incoming/outgoing communication events. The specification tuples are expressed in the ReSpecT language [4]; ReSpecT is logic-based, Turing-equivalent, and makes it possible to express coordination laws in the form of reactions: it

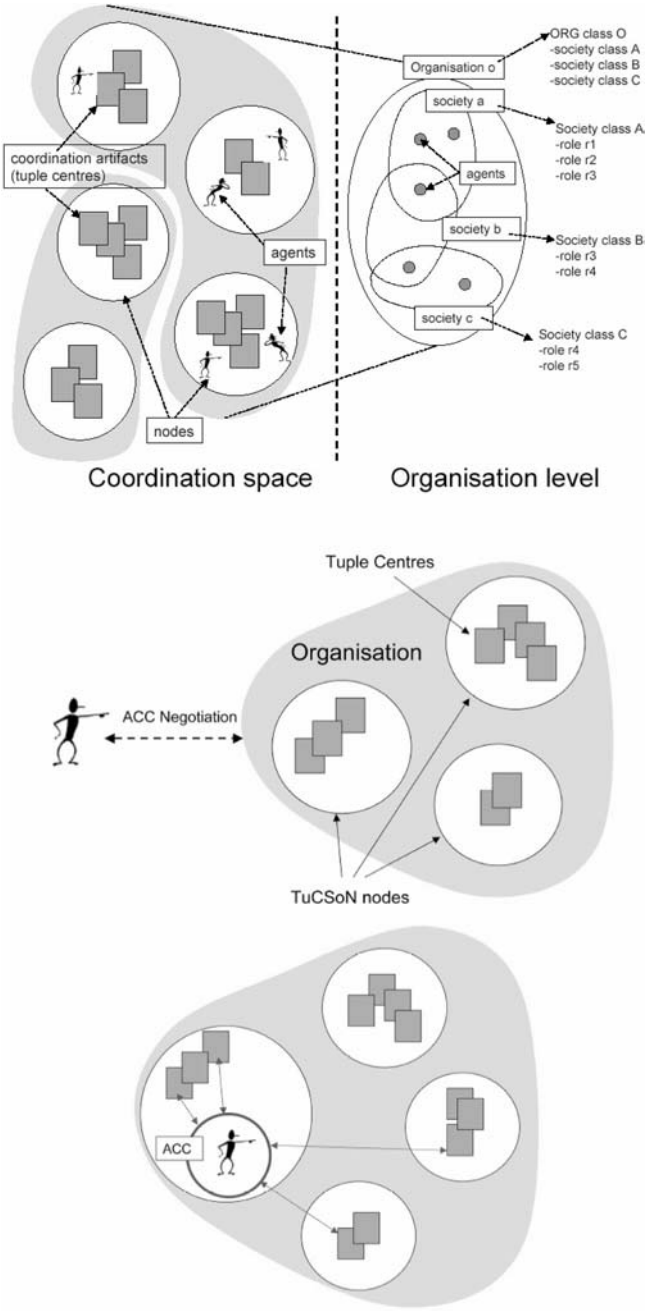


Fig. 1. TuCSoN world. (Top) TuCSoN coordination space, with coordination nodes and tuple centres inside the nodes. (Middle) Negotiation of an Agent Coordination Context (ACC) by an agent that aims at joining an organisation and using its tuple centres. (Bottom) Once inside the organisation, the agent uses the ACC as interface to access tuple centres.

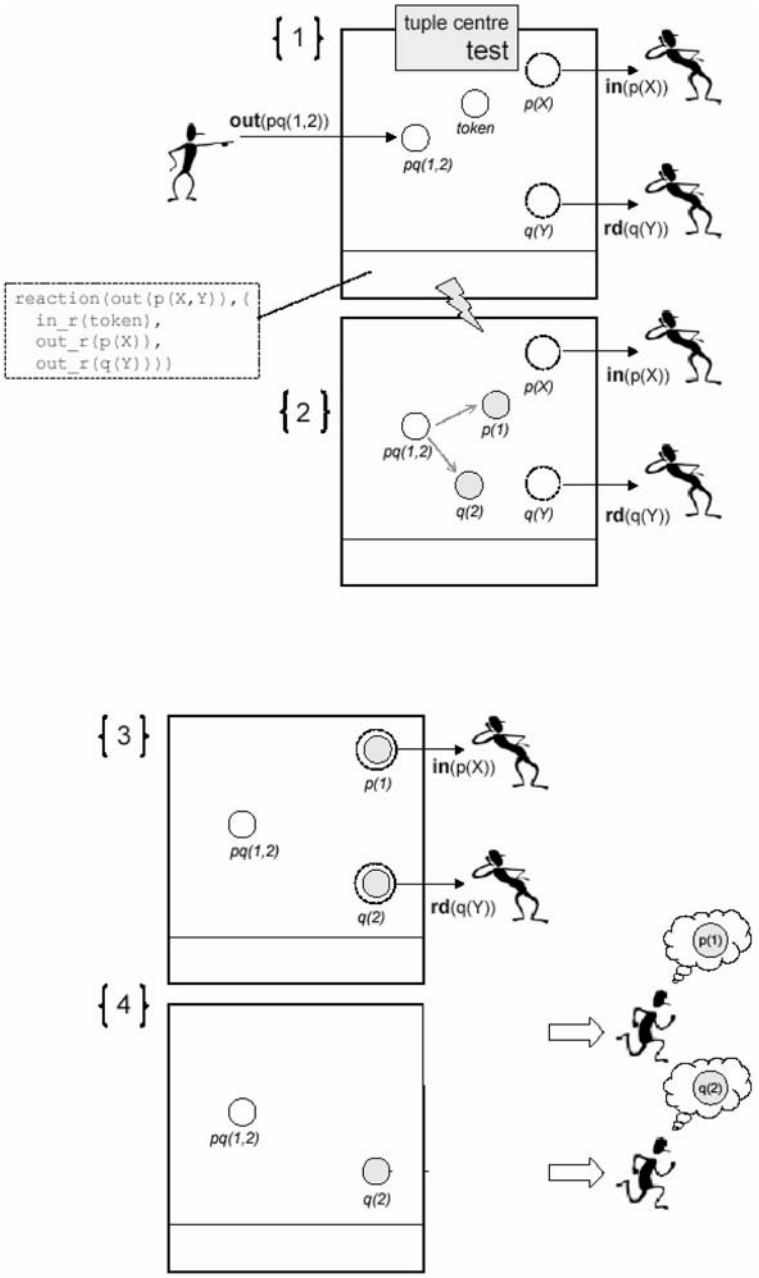


Fig. 2. Using Coordination Artifacts. In the example, three agents interact through the `Test` tuple centre. Four consequent stages are shown: (1) an agent inserts a tuple `pq(1,2)`; (2) the tuple centre reacts according to the reaction shown in the picture, removing the tuple token and inserting the tuples `p(1)` and `q(2)`; (3) and (4) two agents that were waiting for tuples matching the `p(X)` and `q(Y)` templates respectively retrieve and read the tuples just inserted.

can be considered an assembly language for managing interaction and doing coordination. The reactions are triggered by interaction events concerning the tuple centres (both external, such as the invocation of a coordination primitive by agents, and internal, such as the insertion or removal of a tuple by means of a ReSpecT primitive). The semantics of ReSpecT has been formally specified [14], defining the behaviour of the language primitives and the properties of the reaction triggering and execution stages (such as non-determinism and the atomic/transactional behaviour).

So, tuple centres can be seen as general-purpose customisable coordination artifacts, whose behaviour can be dynamically specified, forged and adapted.

From the topology point of view, tuple centres are collected in TuCSoN nodes, spread over the network and belong to specific organisations (see Fig. 1, top). In order to access and use tuple centres of an organisation context, an agent must negotiate and enter an *Agent Coordination Context*, which is used to define its presence / position inside the organisation in terms of allowed actions on the available artifacts (tuple centres) [18]. Fig. 1 shows these two basic stages: first an agent negotiates the configuration of the ACC by means of the services provided by the infrastructure (Fig. 1, middle); then, if the agent request is satisfiable (according to the organisation rules defined by the specific organisation context), an ACC with the specific configuration is created and entered logically by the agent [17] (Fig. 1, bottom).

Summing up, in this section we provided an overview of the agent and coordination artifact abstractions for engineering complex systems. In next section we will consider feature-based systems as complex systems, and then amenable to be engineered using the agent oriented approach.

2 Feature-Based Systems as Complex Systems

Features can be defined as optional-bits of self-contained functionalities, and are used as structuring mechanism for engineering systems. Originally, the concept has emerged in telecommunication context, in particular in telephone systems analysis as a way of describing optional services to which telephone users may subscribe [12].

Engineering a system by means of the feature abstraction involves starting with a basic system core and then completing/extending the systems by adding new features, as self-contained functionalities. Typically features cannot be represented directly neither as objects, nor as components, since they concern functionalities which span on multiple objects/modules/components. We can recognise in the feature approach also a methodological aim: a feature includes a description about how to transform a base system, by adding new encapsulated functionality; the challenge is then the capability to add this functionality which typically affects many parts of the system without rewriting/changing all the system parts.

Feature integration and composition are then one of the main issues, and can be described as the process of adding a feature to a base system. This process is clearly non-monotonic, since a new feature may override behaviour / properties present in the base system and in other features.

From the integration issues, it emerges the main problem challenging research on feature-based systems, that is the *feature interaction problem* [2], when several features are integrated in the same system, they may interfere with each other in undesirable ways. Conversely, in the same integration context, features can work together positively: that is, there is *feature inter-working*, obtaining new functionalities beyond the sum of the individual features. So detecting feature interaction that creates interference and not inter-working is a problem of primary importance. It is worth noting that in other computer science contexts (such as Software Engineering, Artificial Intelligence, Programming Languages) the term *interaction* is used with a different acceptation, not as interference but as the (usually desirable) effect of dependencies among components of a system; in particular, it is used often to indicate communication (even if communication can be considered as a special form of interaction). So, we can devise two different perspectives on the notion of interaction: from one side, each feature emerges from the (positive) interactions of the components of a system: in fact, the functionalities encapsulated by a feature can be often achieved only by suitably coordinating the communications and more generally the dependencies among the components. On the other side, the interaction between features is instead what must be avoided or controlled.

Other interesting issues concerning feature research which are fundamental for the engineering of systems are:

- feature models and languages – defining models and languages for specifying and describing features. Which computational models are useful for capturing feature abstraction? Either using purely algorithmic languages or object-oriented ones seem not to be adequate to capture the real essence of features, as functionalities involving multiple entities. For this reason Aspect Oriented paradigm seems to be more effective; however, this approach seems not to be adequate for handling feature interaction and inter-working issues; moreover, Aspect Oriented focuses mainly at the compile time and hardly provides means to support runtime/dynamic integration/evolution of features. Policy-driven models have been recently considered as another possible paradigm, mapping features as policies that rule interaction components of a system and provide in the overall the functionalities required. Here we claim that also coordination models and languages [6, 20] can be considered interesting for modelling and expressing features. These approaches promote a clear separation between computation from coordination issues, and provide specific abstraction for specifying and enacting the latter; the coordination dimension concerns interaction management, and is not related to a specific component but to the laws that glue components together;
- formal feature – the possibility of defining a formal semantics of feature models and languages is fundamental for the verification of formal properties, in particular to face the feature interaction problem with formal tools;
- feature evolution – engineering systems with some degree of flexibility implies the need of supporting not only the integration of new features, but also the evolution/change/replacement of existing ones, without breaking system consistency;

- feature for concurrent systems – almost all feature-based application scenarios include aspects concerning concurrency and distribution, where features concern components distributed on different execution contexts, on different nodes of a network, so with multiple control flows;
- feature infrastructure – modern application scenarios require also more and more the capability of changing systems behaviour at runtime, dynamically; this implies the capability of integrating, removing, evolving features at runtime, toward runtime feature *plug-and-play* [22, 23]. This capability typically can be faced by means of proper software infrastructures, providing runtime support (services) for manipulating abstractions (features in this case) as first class entities also at runtime.

3 Engineering Features with Agents and Coordination Artifacts Infrastructure

Since agents and coordination abstractions are good for complex systems engineering and feature-based systems provide a high degree of complexity, it is interesting to investigate the application of the former as engineering tool for the latter. Accordingly, this section provides a perspective, a possible way to conceive the engineering of feature within a MAS context, in particular discussing the main points and challenges that concern feature based systems as sketched in the previous section.

As mentioned in Section 1, agent oriented software engineering accounts for identifying in the analysis stage individual and social tasks required to reach system objectives. Individual tasks are mapped onto individual agents while both the social tasks with norms and the global properties of the system are mapped onto agent societies or groups, which adopt some kind of coordination artifacts in order to achieve the social objectives. A feature encapsulates a functionality concerning multiple components inside a system; then it comes natural to recognise the feature concept as a social task, which can be achieved by suitably adapting existing coordination artifacts, and possibly introducing new agents providing specific competencies that the features require. In other words, a new feature implicitly or explicitly involves new relationships and rules among a set of existing components or activities, and possibly the introduction of new specific activities (individual tasks) related to the relationships: so, first the new relationships and rules can be expressed by coordination laws extending the behaviour of the coordination artifact gluing existing individual activities, and then the new individual tasks are assigned to new agents (roles).

Given this conceptual mapping, feature plug-and-play is obtained by evolving dynamically the behaviour of the coordination artifacts, and by possibly introducing new agents, responsible of the execution of some new individual task related to the new feature. If the system is well-engineered, the integration of a new feature would not imply changing existing agents, but only the glue aggregating and coordinating their activities.

So, the feature interaction problem can be faced focusing primarily on the coordination artifacts behaviour, checking for the presence of coordination laws

which interfere each other in an undesirable way. In this framework, feature interaction emerges from dependencies that have been not properly handled in the coordination medium. The same reasoning applies for the opposite issue, that is feature inter-working: possible desirable interaction among features can be here obtained and managed by suitable coordination laws which do not interfere, but work together fruitfully. For all the other issues:

- feature models and languages – in this case the model and language used to define and enact features is strictly related to the coordination model (and language) adopted to support multi-agent system coordination, in particular to define coordination artifacts and their behaviour. For instance, in the case of TuCSoN a feature would be mainly described in terms of coordination laws expressed in the ReSpecT language;
- formal feature – coordination models and languages with formal semantics can be suitable for investigating the verification of formal properties of coordination. This is the case of tuple centre model and ReSpecT language, for instance. In this context, this accounts for verifying formal properties of features, mapped on top of coordination laws enacted by media. This possibility is a key to face the feature-interaction problem from a formal point of view;
- feature evolution – feature evolution is supported in this case by the capability of changing dynamically the behaviour of the coordination artifacts by changing/adapting the coordination laws concerning the feature. In the case of TuCSoN, for instance, the behaviour of a tuple centre can be inspected and changed dynamically by means of suitable tools (for humans) or by coordination primitives that can be used directly by agents;
- feature infrastructure – any model based on agents and coordination artifacts necessarily requires the notion of infrastructure to support at runtime these abstractions and their interactions, by means of suitable services. Then, the infrastructure has a key role in supporting issues related to integration and evolution of features, in particular by providing the services to inspect and change at runtime the behaviour of the coordination artifacts used for implementing them.

4 A Simple Case Study: Email System

In this section we exemplify the approach described in previous section using TuCSoN to engineer a simple case study typically found in feature literature [7, 23], an email service[1]. The objective of our system is to setup an email service, which can be used both from human and artificial agents. The functionalities of the basic system concern receiving, sending and visualising (for humans) emails.

[1] The source code presented in this section can be tested by using TuCSoN technology, available as open source project at http://lia.deis.unibo.it/research/tucson or http://tucson.sourceforge.net

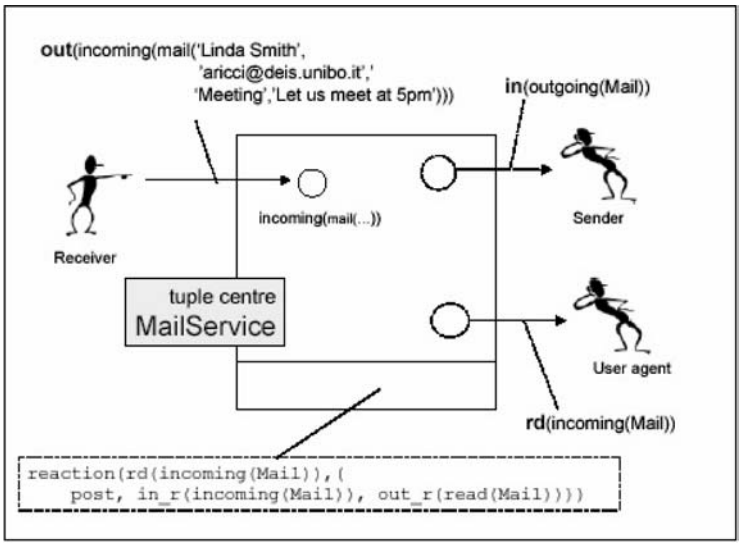


Fig. 3. The basic email service engineered with TuCSoN. Three agents are responsible for individual tasks (sending emails, receiving emails and assisting the human user) and the `MailService` tuple centre is used to support their coordination.

The core service will be then extended by integrating new features, the encryption/ decryption capability and the auto-reply functionality. With a simple analysis, three basic roles can be identified:

- the receiver role, responsible of receiving incoming emails though of the POP protocol/infrastructure;
- the sender role, responsible of sending outgoing emails using of the SMTP protocol/infrastructure;
- the user interface role, responsible of assisting users in checking and visualising incoming emails, and composing outgoing emails.

The social laws glue the activities of these roles fruitfully. The roles are assigned and played by TuCSoN agents, with the capabilities required to fulfill role tasks. In order to coordinate the individual activities, we use a single tuple centre, called `MailService`. An overall picture of the basic system is depicted in Fig. 3. We use the logic tuple

```
mail(Sender,Receiver,Subject,Content)
```

to represent email information, where simple string can be used to describe all the fields of the mail. An example of mail tuple can be:

```
mail('Linda Smith', 'aricci@deis.unibo.it', 'Meeting',
     'Let us meet at 5.00 pm. Bye, Linda')
```

Table 1. ReSpecT code defining the basic coordination behaviour of the `MailService` tuple centre. Basically, in order to avoid that the same incoming mail could be read more than once, when an `incoming` tuple is read by means of the `rd` operation, the tuple is removed and a new tuple `read` with the same mail information is inserted.

```
reaction(rd(incoming(Mail)),(
    post, in_r(incoming(Mail)), out_r(read(Mail)))).
```

The presence of incoming and outgoing emails are represented respectively by tuples `incoming(`*`Mail`*`)` and `outgoing(`*`Mail`*`)`, where *Mail* follows the format described above. Receiver agents notify the arrival of a new email to the agent society by placing a new `incoming` tuple in the `MailService` tuple centre (by means of an `out` operation). Sender agents collect `outgoing` tuples by means of `in` or `inp` operation as they appear in the tuple centre, and use the information to send emails via the SMTP protocol. User agents collect and visualise new incoming emails for a human user by reading `incoming` tuples through the `rd` operation, and also assist a human user in sending emails by inserting `outgoing` tuples with proper data. The synchronisation capabilities provided by the basic coordination primitives (`in`, `out`, etc.) are sufficient for realising the simple coordination required by the basic service core. In addition, we add a coordination law to avoid that incoming emails could be read and visualised more than once, by turning `incoming(`*`Mail`*`)` tuples in `read(`*`Mail`*`)` tuples as soon as the `incoming` tuples are read the first time. In order to enact this coordination behaviour, the tuple centre `MailService` is programmed with the reaction shown in Table 1.

4.1 Handling Feature Integration: Encryption and Decryption

Now we aim at extending the basic system core by integrating the encryption/ decryption feature [7]. Abstracting from details, we suppose that format of the logic tuple deployed for an encrypted email is:

`mail(`*`Sender`*`,`*`Receiver`*`,encrypted(`*`Subject`*`),encrypted(`*`Content`*`))`

In order to add the encryption feature we extend the basic society by adding an agent (role) able to encrypt information, and the coordination laws gluing the new activity with sender one. Basically, the activity of the encryption agent is triggered by the presence of tuples representing mails to encrypt `mail_to_encrypt(`*`Mail`*`)` and produces tuples with mail encrypted `mail_encrypted(`*`EncryptedMail`*`)`. In particular, the encryption agent can get `mail_to_encrypt` tuples by means of `in` operations, and provide the mail encrypted tuples by means of `out` operations. In order to provide this new feature, we extend the behaviour of the tuple centre with the reactions depicted in Ta-

Table 2. ReSpecT code defining the encrypt (*top*) and decrypt (*bottom*) feature.

```
1   reaction(out(outgoing(Mail)),(
        rd_r(encryption(true)), in_r(outgoing(Mail)),
        out_r(mail_to_encrypt(Mail)))).

2   reaction(out(mail_encrypted(Mail)),(
        in_r(mail_encrypted(Mail)), out_r(outgoing(Mail)))).
```

```
3   reaction(out(incoming(mail(S,R,encrypted(Subj),encrypted(C)))),(
        in_r(incoming(mail(S,R,encrypted(Subj),encrypted(C)))),
        out_r(mail_to_decrypt(mail(S,R,encrypted(Subj),encrypted(C)))))).

4   reaction(out(mail_decrypted(Mail)),(
        in_r(mail_decrypted(Mail)), out_r(incoming(Mail)))).
```

ble 2 (top). For hypothesis, the presence of the tuple `encryption(true)` indicates that the encryption feature is active. According to the extended behaviour, when an outgoing non-encrypted email is inserted in the tuple centre and the encryption service is on, the tuple with the information in clear is removed and a `mail_to_encrypt` tuple inserted (reaction 1). Then, a free encryption agent would retrieve the tuple by means of an `in` or an `inp` primitive, do its job and then insert the resulting `mail_encrypted` tuple in the tuple centre. The tuple centre reacts to this action, by removing the `mail_encrypted` tuple and inserting a new `outgoing` tuple, containing the encrypted email (reaction 2). As in the previous case, the `outgoing` tuple would be then collected and sent away by the sender agent.

An analogous approach can be adopted for the decryption feature. As for the encryption, we extend the behaviour with suitable coordination laws (see Table 2, bottom) and introduce a new agent able to decrypt the information (or we can assign the decryption task to the same encryption agent). Analogously to the encryption agent, the activity of the decryption agent is triggered by the presence of tuples representing mails to decrypt `mail_to_decrypt(`*`EncryptedMail`*`)` and produces tuples `mail_decrypted(`*`Mail`*`)` with mails decrypted. When an `incoming` tuple is inserted with an encrypted mail, the tuple is removed and a new tuple `mail_to_decrypt` inserted (reaction 3). Then the decryption agent comes into play, consuming the `mail_to_decrypt` tuple, doing its decryption job and then inserting the new decrypted tuple, `mail_decrypted`. The tuple centre reacts by removing the `mail_decrypted` tuple and inserting a new `incoming` tuple, containing the decrypted email (reaction 4). The `incoming` tuple is then collected by the user agent and properly visualised.

What is important to note here is that the dynamic/runtime integration of the new encryption and decryption feature has not caused the intervention

Table 3. ReSpecT code defining the auto-reply feature (*top*) and code required to avoid feature interaction with the encryption/decryption feature (*bottom*).

```
1   reaction(out(incoming(Mail)),(
        rd_r(auto_reply(true)), out_r(mail_toreply(Mail)))).

2   reaction(out(mail_reply(ReplyMail)),(
        in_r(mail_reply(ReplyMail)), out_r(outgoing(ReplyMail)))).
```

```
3   reaction(out_r(mail_toreply(S,R,encrypted(Subj),encrypted(C))),(
        in_r(mail_toreply(mail(S,R,encrypted(Subj),encrypted(C)))),
        out_r(mail_suspended_reply(mail(S,R,encrypted(Subj),
                                        encrypted(C))))).

4   reaction(out_r(mail_decrypted(mail(S,R,Subj,Cont))),(
        in_r(mail_suspended_reply(mail(S,R,_,_))),
        out r(mail_toreply(S,R,Subj,Cont)))).
```

on existing running agents, but only the runtime extension of the behaviour of coordination artifact with new coordination laws and the introduction of new specialised agents.

4.2 Handling Feature Interaction: Encryption/Decryption plus Auto-reply

Now suppose that we want to add the auto-reply feature, enabling automatic answers to all incoming emails. As remarked in literature, this feature and the encryption/decryption feature interfere [23]. For instance the encryption and auto-responder interact as follows: the auto-responder answers emails automatically by quoting the subject field of the incoming email; if an encrypted email is decrypted first and then processed by the auto-responder, an email with the subject of the email will be returned. This however leaks of the originally encrypted subject if the outgoing email is sent in plain.

As for the encryption and decryption feature, we can integrate the new feature by extending the behaviour of the tuple centre with proper coordination laws (see Table 3, top) and by introducing a new agent able to compose the answer mail from the incoming one. The activity of this auto-responder agent is triggered by the presence of tuples `mail_toreply(`*`Mail`*`)` representing mails to be answered and produces tuples `mail_reply(`*`ReplyMail`*`)` with the replies. When an `incoming` tuple is inserted and the auto-reply feature is active (a tuple `auto_reply(true)` is present in the tuple set), the tuple `mail_toreply` is inserted in the tuple centre (reaction 1). Then the auto-responder agent comes into play, consuming the `mail_toreply`, doing its job – in this simple case creating a reply mail by quoting the subject of the original one – and then insert-

ing the new reply tuple, `mail_reply`. The tuple centre reacts by removing the `mail_reply` tuple and inserting a new `outgoing` tuple, with the reply mail to be sent (reaction 2). The `outgoing` tuple would be finally collected and sent away by the sender agent.

However, it is easy to verify that this solution suffers the feature interaction problem. The problems arise with incoming encrypted emails, which require to receive an encrypted auto-reply. In this case, the auto-responder agent would be asked to create a reply from an encrypted email, and then to be able to decrypt it first. It is worth noting that there are no problems or interference with the outgoing emails: when the auto-reply agent inserts a non-encrypted outgoing reply, it is collected and encrypted by the encryption agent, and then sent away by the sender agent. In order to decrypt the mail the auto-responder agent could use the available decryption service, by inserting a suitable `mail_to_decrypt` and then collecting the tuple `mail_decrypted` as a result. This could be a solution, however in this way we would create a coupling between the auto-responder agent with the decryption service, that is the auto-reply agent must be aware of the existence of encrypted emails and of the related service for their decryption. The problem can be solved by focusing completely on the coordination part, that is adapting the glue among the activities. Our goal is to trigger the decryption activity before the encrypted mail would be collected by the autoresponder agent. To do this we can either modify the reactions added for the auto-reply feature, in order to have a different coordination behaviour if the encryption/decryption feature is active; however this would mean changing the glue related to a feature because of the presence of another feature. Or, we can avoid any change to the existent glue and solving the problem by adding the reactions useful *to manage the dependencies between the coordination laws related to the two features*, by constraining the execution order of the decryption and auto-reply activity. The reactions are shown in Table 3 (bottom). When the auto-reply service is triggered by inserting a `mail_toreply` tuple with an encrypted mail, the tuple centre reacts and the tuple is removed – to avoid the immediate service triggering – and a new `mail_suspended_reply` tuple inserted to keep track of the suspension (reaction 3). Then, when the decryption service finishes and the `incoming` tuple is inserted with the decrypted email, related to a suspended reply, the `mail_suspended_reply` is removed and the auto-reply service is triggered again by inserting the proper `mail_toreply` tuple.

So, what is important to remark here is that the feature interaction could be managed without any intervention on the agents responsible on the individual activities, but only by adapting or, possibly, extending the glue that binds the activities together. Of course, the solution (interaction protocols and reactions) provided in this example are useful only in principle, for reasoning about the problem: real-world systems would require more solid and well-engineered tuple formats, protocols and reactions.

5 Conclusions

In this paper we investigated the relationships between feature based system and agent oriented software engineering, in particular we sketched out a possible way to model and design feature-based systems in terms of agents and coordination artifacts. Some benefits of the approach have been remarked, in particular the support provided by agents and artifacts to feature runtime integration and evolution, and for the management of features in the context of concurrent and distributed systems. The bridge between the two worlds has triggered some interesting challenges coming from the feature context – such as the feature interaction problem: we found these issues very interesting and challenging also using agents for the system engineering, and should be subject of future investigations.

Acknowledgments

I would like to thank Andrea Omicini, whose support and ideas have been fundamental first for participating to the Dagstuhl event and then for conceiving this article.

References

1. F. M. T. Brazier, B. M. Dunin-Keplicz, N. R. Jennings, and J. Treur. DESIRE: Modelling multi-agent systems in a compositional formal framework. *International Journal of Cooperative Information Systems*, 6(1):67–94, 1997.
2. M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature Interaction: a critical review and considered forecast. *Computer Networks*, (41):115–142, 2003.
3. P. Ciancarini, A. Omicini, and F. Zambonelli. Multiagent system engineering: the coordination viewpoint. In N. R. Jennings and Y. Lespérance, editors, *Intelligent Agents VI — Agent Theories, Architectures, and Languages*, volume 1767 of *LNAI*, pages 250–259. Springer-Verlag, Feb. 2000.
4. E. Denti, A. Natali, and A. Omicini. On the expressive power of a language for programming coordination media. In *Proc. of the 1998 ACM Symposium on Applied Computing (SAC'98)*, pages 169–177. ACM, 1998. Track on Coordination Models, Languages and Applications.
5. L. Gasser. Mas infrastructure: Definitions, needs, and prospects. In T. Wagner and O. Rana, editors, *Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems*, volume 1887 of *LNAI*. Springer-Verlag, 2001.
6. D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, Feb. 1992.
7. R. J. Hall. Feature interactions in electronic mail. In *IEEE Workshop on Feature Interaction*. IOS-press, 2000.
8. C. Iglesias, M. Garijo, and J. Gonzalez. A survey of agent-oriented methodologies. In J. Müller, M. P. Singh, and A. S. Rao, editors, *Proceedings of the 5th International Workshop on Intelligent Agents V : Agent Theories, Architectures, and Languages (ATAL-98)*, volume 1555, pages 317–330. Springer-Verlag: Heidelberg, Germany, 1999.

9. C. Iglesias, M. Garijo, J. Gonzalez, and J. R. Velasco. Analysis and design of multiagent systems using MAS-common KADS. In *Agent Theories, Architectures, and Languages*, pages 313–327, 1997.
10. N. Jennings and S. Bussmann. Agent-based control systems: Why are they suited to engineering complex systems? *Control Systems Magazine*, 23(3):61–73, June 2003.
11. N. R. Jennings and M. Wooldridge. Agent-oriented software engineering. In F. J. Garijo and M. Boman, editors, *Multi-Agent Systems Engineering*, volume 1647 of *LNAI*. Springer-Verlag, 1999. 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'99), Valencia (E), 30 June – 2 July 1999, Proceedings.
12. D. Keck and P. Kuehn. The feature and service interaction problem in telecommunications systems: a survey. *Transaction on Software Engineering*, 24(10):779–796, Oct. 1998.
13. A. Omicini. SODA: Societies and infrastructures in the analysis and design of agent-based systems. In P. Ciancarini and M. J. Wooldridge, editors, *Agent-Oriented Software Engineering*, volume 1957 of *LNCS*, pages 185–193. Springer-Verlag, 2001.
14. A. Omicini and E. Denti. Formal ReSpecT. In A. Dovier, M. C. Meo, and A. Omicini, editors, *Declarative Programming – Selected Papers from AGP'00*, volume 48 of *Electronic Notes in Theoretical Computer Science*, pages 179–196. Elsevier Science B. V., 2001.
15. A. Omicini and E. Denti. From tuple spaces to tuple centres. *Science of Computer Programming*, 41(3):277–294, Nov. 2001.
16. A. Omicini and S. Ossowski. Objective versus subjective coordination in the engineering of agent systems. In M. Klusch, S. Bergamaschi, P. Edwards, and P. Petta, editors, *Intelligent Information Agents: An AgentLink Perspective*, volume 2586 of *LNAI: State-of-the-Art Survey*, pages 179–202. Springer-Verlag, Mar. 2003.
17. A. Omicini and A. Ricci. Reasoning about organisation: Shaping the infrastructure. *AI*IA Notizie*, XVI(2):7–16, June 2003.
18. A. Omicini, A. Ricci, and M. Viroli. Formal specification and enactment of security policies through Agent Coordination Contexts. In R. Focardi and G. Zavattaro, editors, *Security Issues in Coordination Models, Languages and Systems*, volume 85(3) of *Electronic Notes in Theoretical Computer Science*. Elsevier Science B. V., Aug. 2003.
19. A. Omicini and F. Zambonelli. Coordination for Internet application development. *Autonomous Agents and Multi-Agent Systems*, 2(3):251–269, Sept. 1999. Special Issue: Coordination Mechanisms for Web Agents.
20. G. A. Papadopoulos and F. Arbab. Coordination models and languages. *Advances in Computers*, 46(The Engineering of Large Systems):329–400, Aug. 1998.
21. A. Perini, P. Bresciani, F. Giunchiglia, P. Giorgini, and J. Mylopoulos. A knowledge level software engineering methodology for agent oriented programming. In *Proceedings of the International Conference on Autonomous Agent (Agents '01)*, May 2001.
22. M. Plath and M. Ryan. Plug-and-play features. In K. Kimbler and L. G. Bouma, editors, *Feature Interactions in Telecommunications and Software Systems*, pages 150–164. IOS-press, 1998.
23. C. Prehofer. Plug-and-play composition of features and feature interactions with statechard diagrams. In *IEEE Workshop on Feature Interaction*. IOS-press, 2003.

24. A. Ricci, A. Omicini, and E. Denti. Activity Theory as a framework for MAS coordination. In P. Petta, R. Tolksdorf, and F. Zambonelli, editors, *Engineering Societies in the Agents World III*, volume 2577 of *LNCS*, pages 96–110. Springer-Verlag, Apr. 2003.
25. M. Viroli and A. Omicini. Coordination as a service: Ontological and formal foundation. In A. Brogi and J.-M. Jacquet, editors, *FOCLASA 2002 – Foundations of Coordination Languages and Software Architecture*, volume 68(3) of *Electronic Notes in Theoretical Computer Science*. Elsevier Science B. V., Mar. 2003. CONCUR 2002 Satellite Workshop, 19 Aug. 2002, Brno, Czech Republic, Proceedings.
26. M. Wooldridge and P. Ciancarini. Agent-Oriented Software Engineering: The State of the Art. In P. Ciancarini and M. Wooldridge, editors, *First Int. Workshop on Agent-Oriented Software Engineering*, volume 1957, pages 1–28. Springer-Verlag, Berlin, 2000.
27. M. Wooldridge and P. Ciancarini. Agent-oriented software engineering: The state of the art. In *Handbook of Software Engineering and Knowledge Engineering*. World Scientific Publishing, 2001.
28. M. Wooldridge, N. R. Jennings, and D. Kinny. The gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.
29. M. J. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.
30. F. Zambonelli, N. R. Jennings, A. Omicini, and M. Wooldridge. Agent-oriented software engineering for Internet applications. In A. Omicini, F. Zambonelli, M. Klusch, and R. Tolksdorf, editors, *Coordination of Internet Agents*, chapter 13, pages 369–398. Springer-Verlag, Mar. 2001.

Author Index