Christine Choppy
Oleg Sokolsky (Eds.)

Foundations of Computer Software

Future Trends and Techniques for Development

LNCS 6028

15th Monterey Workshop 2008
Budapest, Hungary, September 2008
Revised Selected Papers

Springer

Lecture Notes in Computer Science 6028

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

Christine Choppy Oleg Sokolsky (Eds.)

Foundations of Computer Software

Future Trends and Techniques for Development

15th Monterey Workshop 2008
Budapest, Hungary, September 24-26, 2008
Revised Selected Papers

Springer

Volume Editors

Christine Choppy
Laboratoire d'Informatique de l'Université Paris Nord
UMR CNRS 7030, Institut Galilée
99 Avenue J.B. Clément, 93430 Villetaneuse, France
E-mail: Christine.Choppy@lipn.univ-paris13.fr

Oleg Sokolsky
University of Pennsylvania
Department of Computer Science and Information Science
3300 Walnut Street, Philadelphia, PA 19104-6389, USA
E-mail: sokolsky@cis.upenn.edu

Library of Congress Control Number: 2010924335

CR Subject Classification (1998): H.4, H.3, C.2, D.2, H.5, C.2.4

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-642-12565-4 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-12565-2 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

Preface

This volume contains revised and expanded versions of the papers presented at the 15^{th} Monterey Workshop, held during September 24–26, 2008 in Budapest, Hungary.

The Monterey Workshops series was initiated in 1993 by Dr. David Hislop, a longtime program manager at the U.S. Army Research Office, with the purpose of exploring the critical problems associated with cost-effective development of high-quality software systems. During their 15-year history, the Monterey Workshops have brought together scientists that share a common interest in software development research serving practical advances in next-generation software-intensive systems. Each year is dedicated to a particular topic of critical importance. In recent years, workshop topics were "Innovations for Requirement Analysis: From Stakeholders Needs to Formal Designs" (2007 in Monterey, California), "Composition of Embedded Systems, Scientific and Industrial Issues" (2008 in Paris, France), "Networked Systems: Realization of Reliable Systems on Unreliable Networked Platforms" (2005 in Laguna Beach, California), "Software Engineering Tools: Compatibility and Integration"(2004 in Vienna, Austria), "Engineering for Embedded Systems: From Requirements to Implementation" (2003 in Chicago, Illinois), "Radical Innovations of Software and Systems Engineering in the Future" (2002 in Venice, Italy).

The topic of the 2008 workshop was "Foundations of Computer Software, Future Trends and Techniques for Development." Modern computer systems manage very large amounts of information, performing complex computations in a distributed way. At the same time, there is a need to display information in a way that aids human actors in the interpretation of this information and in decision making. The systems are becoming more and more dynamic, adaptive, and increasingly pervasive. Computer systems affect all aspects of our lives, from the electric grid to banking to drive- and fly-by-wire control. Safety and security are becoming central issues of computer system design, and verification, validation, and certification are critical. Hardware technology and user needs are both changing faster than software development methods evolve to accommodate them.

There is a growing concern in the research community that existing foundations of software development are not adequate for this new dynamic world and that software development is becoming increasingly ad hoc. The papers presented at the workshop explore how the foundations and development techniques of computer software could be adapted to address such a challenge. Material presented in the papers spans the whole software life cycle, starting from specification and analysis, design and the choice of architectures, large-scale, real-world software development, code generation and configuration, deployment, and evolution.

We are grateful to the Steering Committee, the Local Organizing Committee, and the invited speakers for making the workshop a success. We acknowledge generous sponsorship from the U.S. Army Research Office (Dr. Purush Iyer) and from the U.S. Air Force Office of Special Research (Dr. David Luginbuhl).

This volume was prepared using the EasyChair conference management system.

January 2010

Christine Choppy
Oleg Sokolsky

Organization

General Chairs

Janos Sztipanovits	Vanderbilt University, USA
Tadeusz P. Dobrowiecki	Technical University of Budapest, Hungary

Program Chairs

Christine Choppy	Université Paris Nord, France
Oleg Sokolsky	University of Pennsylvania, USA

Steering Committee

Egidio Astesiano	University of Genova, Italy
Manfred Broy	Technical University Munich, Germany
Hermann Kopetz	Vienna University of Technology, Austria
Fabrice Kordon	University of Pierre & Marie Curie, Paris, France
Luqi	Naval Postgraduate School, USA
Zohar Manna	Stanford University, USA
Janos Sztipanovits	Vanderbilt University, USA

Local Organization

Tadeusz Dobrowiecki	Technical University of Budapest, Hungary

Invited Speakers

Radu Calinescu	Oxford University, UK
Claudiu Farcas	University of California, San Diego, USA
Serge Haddad	ENS Cachan, France
Reinhard von Hanxleden	Christian-Albrechts-Universität, Germany
Anne E. Haxthausen	Technical University of Denmark, Denmark
Rolf Hennicker	Ludwig-Maximilians-Universität, Germany
Liviu Iftode	Rutgers University, USA
Ethan K. Jackson	Microsoft Research, USA
Zsolt Kocsis	IBM, Hungary
László Lengyel	Budapest University of Technology and Economics, Hungary
Tom Maibaum	McMaster University, Canada
Bertrand Meyer	ETH Zurich, Switzerland

Peter M. Musial	Naval Postgraduate School, USA
András Pataricza	Budapest University of Technology and Economics, Hungary
Wolfgang Pree	University of Salzburg, Austria
Awais Rashid	Lancaster University, UK
Gianna Reggio	Università di Genova, Italy
Bernhard Rumpe	TU Braunschweig, Germany
Wolfram Schulte	Microsoft Research, USA

Table of Contents

Revising the UML Collaborations: A Well-Founded Approach

Egidio Astesiano and Gianna Reggio

DISI, Università di Genova, Italy
{astes,gianna.reggio}@disi.unige.it

Abstract. We first argue that in some software development areas the need emerges of modelling structural and behavioural aspects of a community of objects cooperating to achieve a specific purpose, say a cooperation, for short. The notion of cooperation is formalized, with a first citizenship status, in the UML 2, as a collaboration. There are however some unclear and problematic spots both on some syntactic and semantic aspects of the UML collaboration. The main goal of this paper is to present first a much simplified metamodel for defining a collaboration, still producing the same notation, with an associated semantics. Rather surprisingly different useful semantic interpretations may be given and are discussed.

1 Introduction

In some areas of software development, quite frequently the need emerges of modelling structural and behavioural aspects of "a community of objects cooperating to achieve a specific purpose"; for the moment we use the word "cooperation" to qualify informally those aspects. We have been led ourselves to experience that need when we have started to address the problem of moving from Business scenarios to SOA solutions, following the request coming from some companies, with which we are collaborating within a joint laboratory. To provide context and motivation we will present two paradigmatic and well-known examples.

The first, and currently perhaps the foremost example can be seen in *SoaML* [1], the current final OMG[1] proposal for a UML profile for *Service Oriented Architectures*. There, the need for modelling a community of objects arises in three key cases. First in dealing with the high level view of a community of participants providing and consuming services, that is modelled by a community service architecture; second, in specifying the architecture of a particular participant, with its subparticipants and possibly external collaborators interacting through service contracts; finally the specification of a *service*, involving its participants and their obligations, is defined by a service contract acting as a binding contract. All three concepts are modelled using the UML collaborations.

Another related area where cooperation is a key concept is *Business Modelling* (see, e.g., [2] for a detailed presentation). A Business Model consists of

[1] www.omg.org

C. Choppy and O. Sokolsky (Eds.): Monterey Workshop 2008, LNCS 6028, pp. 1–23, 2010.

business processes, involving the business entities and their mutual relationships; for each process we need to model the static view of the cooperation among the roles interpreted by the business entities, and the behavioural view showing the dynamic process behaviour; finally we need to give an overall view of the processes constituting the business, the business roles and their relationships.

The concept of cooperation, as we have seen in the case of SoaML, is usually modelled using UML collaboration. Contrary to UML 1, in UML 2 collaboration has become a first class citizen construct, with some clear improvements w.r.t. UML 1, but also with some problematic points especially related to the semantic and structural aspects of the behavioural parts. Moreover, as it often happens with other UML concepts, it is quite difficult to answer questions about the legal use of related aspects by looking at the OMG specification documents [3] ("the arcane details" in the words of [4]). Indeed, more generally, it is extremely difficult to come out with an explicit metamodel related to a construct, due to the very nature of the OMG specification built over many levels of indirect references and specializations (you may easily count up to 30 levels of specialization!) and this raises a serious problem to a developer, as it is recognized by the UML inventors, who in [5] suggest that a developer builds his/her own, usually simplified, conceptual model of the constructs of interest. Their suggestion is confirmed and reinforced by our own experience in projects and SE courses using the UML. Hence, we have decided to come out with a simplified metamodel dealing with the most desirable and useful features, cutting down the almost inextricable complication of too many indirect references. We adopt that approach also dealing with the concept of "cooperation", proposing a new presentation of the UML 2 collaboration, that allows us to deal in a more direct way with the related semantic and structural aspects.

We address the problems we have mentioned in the spirit and within the context of the approach to software development, that we have called "well-founded-methods" (see [6]), hence the title of this paper.

Indeed, in [6] we have argued how the purist attitude of the Formal Method community, to which we have contributed for long time, was not adequate for the current practice of software development. There we provided a detailed motivation of that statement. But here, for lack of room, let us summarize our motivation reporting from that paper just two quotations. The first, by B. Meyer in [7], dismisses an ill-defined dichotomy : "For some scientists, software development is a branch of mathematics; for some engineers it is a branch of applied technology. In reality, it is both." The second comes from C. Jones in [8]: "I assume that the purpose of developing formal methods is to influence practical engineering of computer systems ... It is a measure of my unease with some research in the area of computer science that I feel necessary to state this fact." In that spirit, we have introduced the name "well-formed development methods", roughly meaning a revisitation or possibly a proactive proposal of engineering best practice methods, but with the guarantee that the notation is amenable to a rigorous formal foundation, though such formalization is not apparent to the user.

Returning now to the central theme of this paper, to go "well-founded" implies two directions. The first is to address the cooperation issue within a notational context easy to understand for software engineers. In this respect we feel obliged to stick to the UML notation as close as possible. On the other side, contrary to what often happens when using the UML, we try to rely on a solid foundation, and thus we try to provide a clear syntax in a way that it is easily checkable by a developer, and moreover to define unambiguously the semantics or, at least, to outline the possible different semantics.

These are the main aims of the paper, developed in Sect. 3, after showing as a preliminary in Sect. 2 how the concept of cooperation arises and is currently handled in the areas mentioned in the beginning. A fallout of this work is a simplified UML metamodel which is used as a context for presenting the new collaboration and is sketched in Appendix A.

Though the semantic issues posed by the UML have been addressed in many papers, the semantics of collaborations has not received much attention, to our knowledge, perhaps because collaboration is practically a new concept in the UML 2. There has been some work on the collaboration diagrams of the UML 1, see, e.g., [9], but they were a special form of behavioural diagram very similar to the sequence diagrams, while collaborations are structural diagrams with which a behaviour of various kinds may be associated.

On the other end many proposals for the semantics of the UML 2 not restricted to just one kind of diagram do not consider the collaboration at all. For example the quite extended work of the now terminated "UML 2 Semantics Project"[2] (see [10] for an overview of the results) does not cover the collaboration.

Very recently, in [11], we find a notable attempt at providing an overall semantic framework for relating the different types of diagrams, represented as a "heterogeneous institution environment". So, we have, for example, an Institution of Static Structure, and an Institution of Interaction. The spirit is, indeed, that each behavioural diagram should be treated in a specific institution. But, collaborations are not mentioned, nor it is easy to guess how they could be treated; moreover the problem of the possible modalities of interpreting a behaviour, as outlined later, is clearly beyond the scope of that paper.

2 The Need for Modelling Cooperation

The purpose of this section is twofold. First we show how naturally the concept of cooperation arises in the modelling phase of software development. To this end we consider two relevant examples: modelling Business Processes and adopting a SOA (Service Oreinted Architecture) approach. In both cases we see that cooperation can be intuitively, albeit roughly, modelled with the UML concept of cooperation. Second, by introducing the concept of collaboration and its use in a signicant applicative context, we pave the way to the following section, where some problems associated with the concept of UML collaboration are discussed and to some extent solved.

[2] `http://research.cs.queensu.ca/ stl/internal/uml2/index.html`

2.1 The Business Process Modelling Case

Business Modelling (BM) is a real buzzword these days, with a growing offer of supporting software tools by major and minor vendors. However BM may convey different meanings. In the domain of Business Process Re-engineering and Enterprise Architecture, BM is focused on analyzing, organizing and managing the business activities. On the software development side, BM is viewed as preliminary to and integrated with the development of information systems and is concerned with the alignment of business processes and IT. Anyway, whatever the focus, there seems to be a common consensus for the need or the relevance, at least, of a supporting notation. Still such consensus has not been reached about the choice of a notation, though currently especially BPMN (Business Process Modeling Notation)[3], but also (part of) the UML are the most widely used. Here, we argue about the need of modelling cooperation also in this domain and we refer to a recent paper by us ([2]) for more detailed presentation of the aspects sketched here and also for references. In [2] we extended the method MARS[4] [12,13] to model a Business with its related Business Processes.

We can model the roles for the business entities (business workers, business objects and external systems) in a business process with a UML collaboration, e.g., as in Fig. 1; the various roles are typed by classes defined in a class diagram part of the whole model of the business. Then, the process behaviour, that defines how the business process is carried out, may be modelled by an activity diagram giving a workflow view, where the actions refer to the collaboration participants, see, for example, Fig. 2.

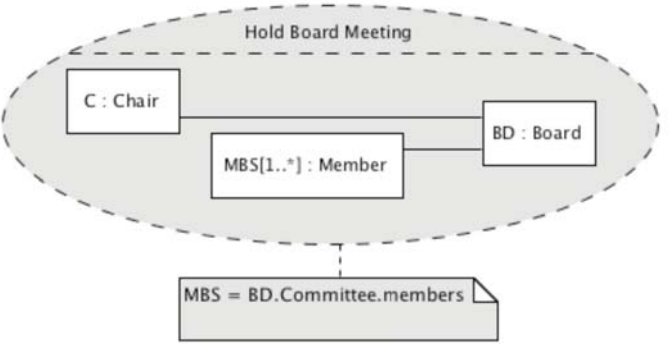


Fig. 1. MARS Business Modelling: participants of a business process

Fig. 3, instead, shows a scenario of the whole business, i.e., a possible case where there are various instances of different business processes running simultaneously and various instances of business entities playing different roles in

[3] `http://www.bpmn.org/`

[4] MARS (Model-based Adaptively Rigorous Software development) is a UML based Model-driven Adaptively Rigorous approach to Software development. MARS falls into the class of what we have called well-founded methods.

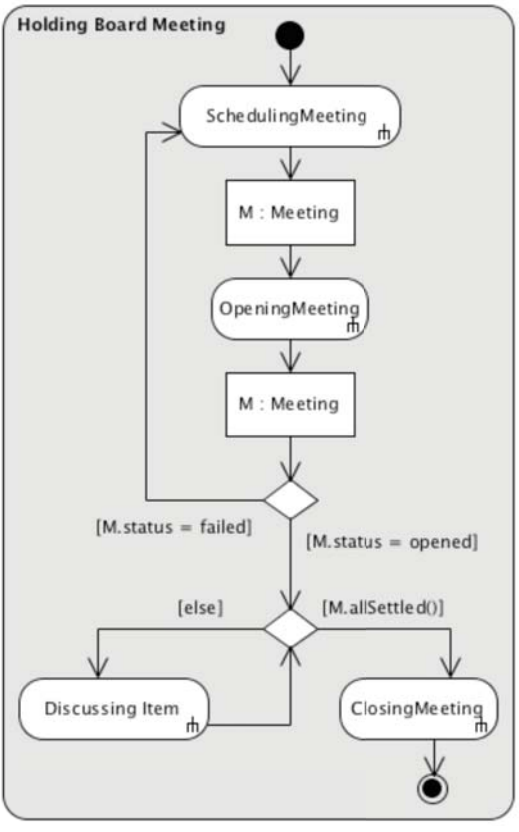


Fig. 2. MARS Business Modelling: wokflow view of a business process

different processes. In the picture we can see, e.g., that there may be two instances of the business process Hold Board Meeting, referring to two different associations, running simultaneously, and that the same person may be the treasurer of two different associations. The instances of the business processes are modelled by means of *collaboration uses*, where the collaboration roles are bound to instance specifications.

2.2 The SoaML Case

SoaML [1] is the current OMG proposal for a UML profile for software development according to the *Service Oriented Architecture* (SOA) paradigm. Remarkably, in the various versions of SoaML we have witnessed a shift towards a high-level view of SOA. Indeed SOA is intended as "a way of organizing and understanding organizations, communities and systems to maximize agility, scale and interoperability. SOA, then, is an architectural paradigm for defining how people, organizations and systems provide and use services to achieve results."

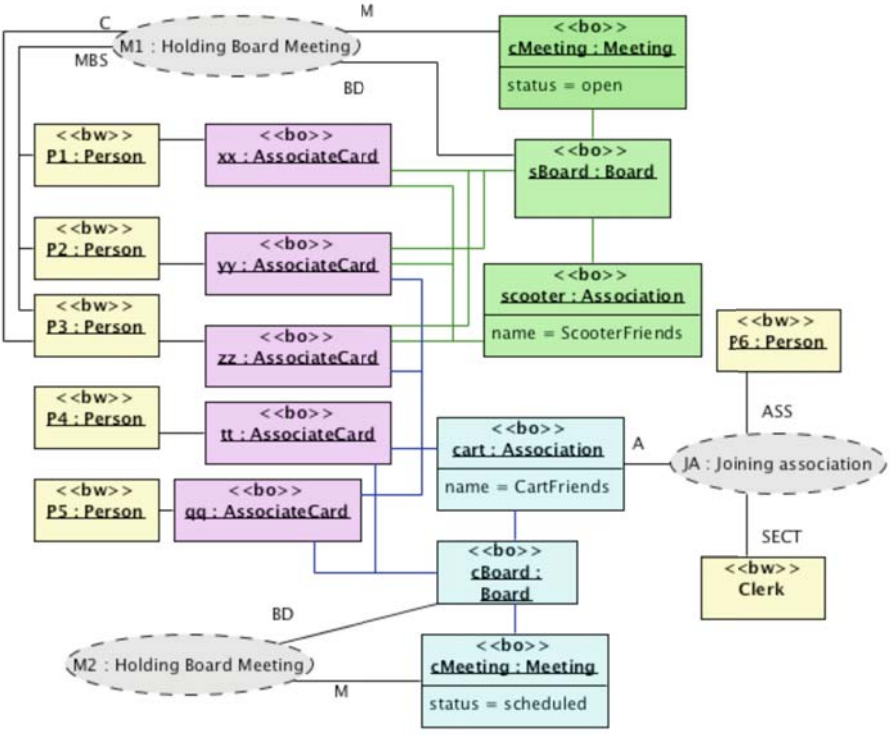


Fig. 3. MARS Business Modelling: business process scenario

Thus, quite naturally, what we have called "cooperation" plays a dominant role within the proposed profile.

Obviously, SoaML being a UML profile, cooperation is modelled by means of UML collaborations, that is used in three key concepts of the services architecture. First, the specification of a service is provided by a *service contract*. "A service is an offer of value to another through a well-defined interface and available to a community (which may be the general public). A service results in work provided to one by another." A service contract defines the terms, conditions, interfaces and choreography that interacting participants must agree to (directly or indirectly) for the service to be enacted – the full specification of a service which includes all the information, choreography and any other "terms and conditions" of the service. Consequently, rather naturally a service contract is modelled by a UML collaboration with an associated behaviour. The static structural part of that collaboration shows the participants (roles) of the services: provider, consumer, and others.

In Fig. 4 the two participants are typed by means of two interfaces stereotyped by ≪ServiceInterface≫, whereas the collaboration is stereotyped by ≪ServiceContract≫; the connector between the two roles depicts that they will exchange messages.

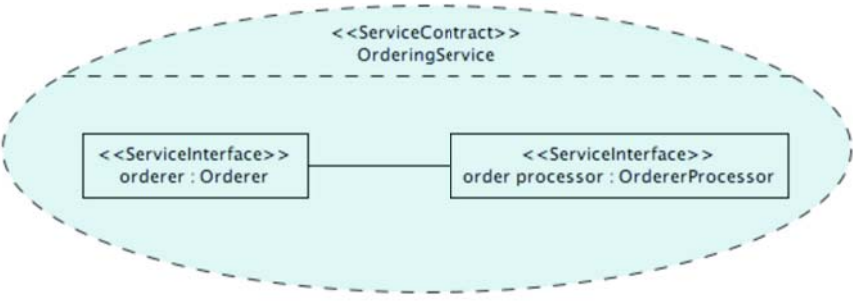


Fig. 4. Example of service contract in SoaML

The behavioural part exposes the choreography of the service, that defines what happens between the provider and consumer participants without defining their internal structure (that have to be compatible with their service contracts).

Notice that here, in SoaML, in principle a behaviour may have any UML form: interaction (e.g., sequence diagrams), state machines, activity diagrams, timed diagrams, ... See Fig. 5 for an example of specification by interaction.

The other use of collaboration in SoaML is in modelling a *services architecture*, which is a network of participant roles providing and consuming services to fulfill a purpose. The services architecture defines the requirements for the types of participants and service realizations that fulfill those roles.

A services architecture is modelled at two levels of granularity. The *Community Services Architecture* is a "top level" view of how independent participants work together for some purpose and is modelled as collaboration stereotyped by ≪ServicesArchitecture≫.

Fig. 6 shows that there are three participants cooperating by means of three services; the fact that two participants are related by a service is represented by a

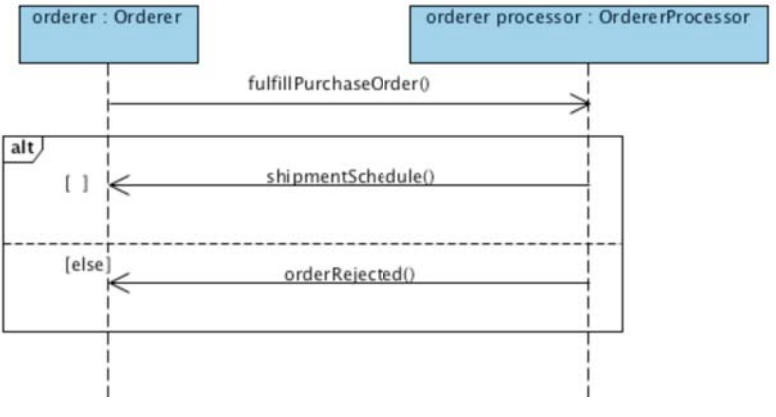


Fig. 5. Example of service choreography in SoaML

collaboration use of the service contract. The lines connecting the collaboration icons to the participant depict who is playing the various roles in the service contracts (technically they are called bindings). Also in a service architecture there may be the specification of a behavioural part representing a business process defining the organization of tasks providing the values which are the goal of the architecture.

The *Participant Architecture* is modelled again as a collaboration stereotyped by ≪ParticipantArchitecture≫, and specifies the internal architecture for a particular participant. In fact, a participant may also have a services architecture, showing how parts of that participant, sub-participants and external collaborators, work together to provide the services owned by the participant. Notably, both views, Services Architecture and Participant Architecture, may be used as a binding specification or as an optional view.

3 Collaboration Revised

3.1 Problems with UML Collaboration

As we have already mentioned, cooperation is modelled in the UML by the model element collaboration. While in the previous versions, collaboration was a rather fuzzy concept, used mainly for helping the intuition, in the UML 2 it has reached a first citizenship status, with a great definitional improvement. However there are still a number of problematic aspects. On the positive side we may note a neat separation between static structure and behaviour; indeed its structure is classified as a composite structure where the parts are roles, the roles of the participants, linked by connectors. Moreover collaboration may be nested by means of the "collaboration use". However the problematic aspects outweigh, in our opinion, the positive ones, for an effective use in software development.

The first problems concern the syntax, that in the UML is defined by metamodelling. As for other model elements, it is quite difficult to check the syntax

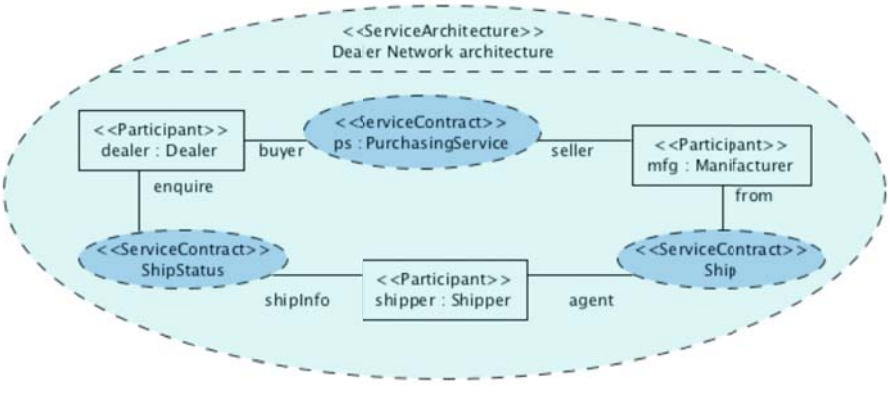


Fig. 6. Example of service architecture in SoaML

against the official OMG specification of the UML. Indeed, that specification has been organized in a hierarchical splitting of concepts (model elements), dictated exclusively by the opportunity of permitting a flexible, highly reusable modularization of the supporting tools, and not much suitable for helping a software developer. Thus it is extremely difficult to come out with an explicit metamodel related to a construct. That is why, oddly enough, the UML inventors speak in [4] of "the arcane details" of the OMG official specification and in [5] suggest a developer to build his/her own, usually simplified, conceptual model of the constructs of interest. Collaborations being a composite structure, the situation is even worse than for other model elements. Experimentally, in some attempts at checking the admissibility of some syntactic parts of a collaboration, we have often counted up to 30 levels of indirect hierarchical references. We will try to eliminate most of those problems by proposing a new metamodel which allows to use all element models of real use in software development practice, while producing the same elements as in the UML. A choice, this last, dictated by the widespread use of the UML, at least for some of its catching visual aspects.

A second source of problems concerns the semantics, especially of the associated behaviour. Notably, there is not even consensus on the type of behavioural diagrams to be used for collaborations. In principle, and as is suggested in the OMG SoaML [1], any kind of behaviour is admitted; but in the OMG UML specification [3] and also in [5] it is suggested that only the use of interaction diagrams is suitable. But the big problems come from the interpretations of the associated behaviour. As we will show, different modalities are possible. Moreover, as the things are, it is totally unclear how behaviours are related in a layered collaboration and the same problem arises for collaboration specialization. Our conceptualization should help also in this respect, but we have first to present a formal discussion of the mentioned issues, before understanding the possible solutions.

3.2 A New Conceptual Model for Collaboration

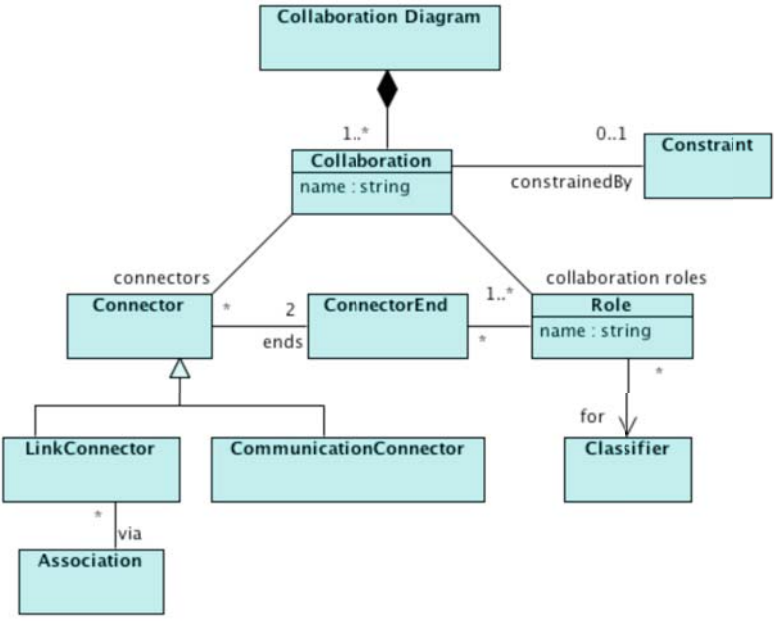
We present in Fig. 7 the fragment of our conceptual metamodel concerning the collaboration. The structure of the models and the class diagrams are shown in Fig. 10 and 11 in the Appendix.

In this paper to simplify the UML class diagrams we follow the convention that the association multiplicity 1 is omitted.

At the diagrammatic level we prefer to speak of *Collaboration (overview) diagram* instead of using the UML term "Composite Structure Diagram", because the last one is rather generic and, it is said, may be used for two different aims: collaboration use and structured classes (but not a simple collaboration).[5]

The metamodel fragment of Fig. 7 allows to define all basic collaborations; for example the ones depicted in Fig. 1 and 4.

[5] From [3, page 191]: "A composite structure diagram depicts the internal structure of a classifier, as well as the use of a collaboration in a collaboration use."

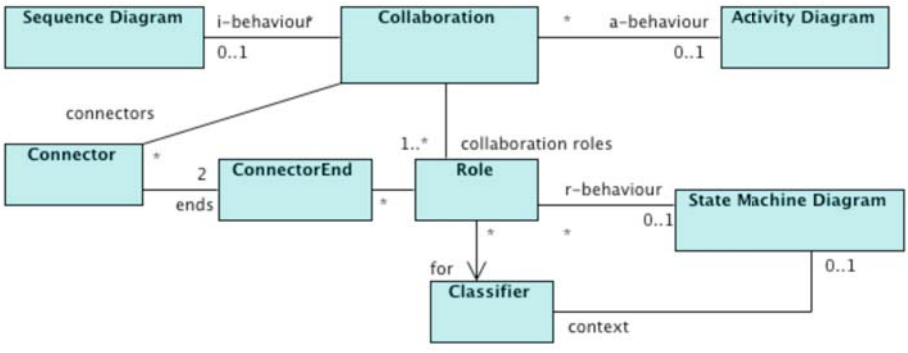


Static Semantics
The connectors may only connect roles of the collaboration to which they belong
A classifier typing a role may be a class or an interface
An association typing a connector must have the two ends typed as the connector ends
The classifiers and the associations typing parts of a collaboration must belong to the model to which the collaboration belongs
The expression defining the constraint, if any, must be correct w.r.t. the environment consisting of the collaboration roles

Fig. 7. Collaboration: conceptual (meta)model

A collaboration is a BehavioredClassifier, that means it may have associated a behaviour. Behaviour in the UML metamodel is a metaclass encompassing many different constructs and quite hard to grasp in its full generality, thus we prefer to define the behaviour of a collaboration in a more specific and direct way. We assume that the behaviour of a collaboration may be defined either by sequence diagrams, or activity diagrams or state machines. The sequence diagram case is quite clear: the objects playing the collaboration roles should be able to perform the message exchanges represented by the sequence diagram; whereas in the case of an activity diagram the objects playing the collaboration roles should be able to perform the actions part of the shown activity respecting the causual relationships modelled by the activity diagram. A state machine should instead be relative to a specific role, and should model the expected behaviour of an object playing such role.

In Fig. 8 we present the fragment of our conceptual (meta)model concerning the behavioural aspect of collaborations; there i-behaviour, a-behaviour and r-behaviour stand respectively for interaction, activity and role behaviour.



Static Semantics
The actions of an a-behaviour must have the form of an operation call over a role of the associated collaboration (including also creation and destruction)
The lifelines of an i-behaviour of a collaboration must be the roles of the same, and if there is a message between two lifelines, there should be a connector in the collaboration between the two associated roles

Fig. 8. Collaboration with associated behaviour: conceptual (meta)model

3.3 (Formal) Semantics of the UML: Our Setting

The formal semantics of collaboration should be given as part of an overall semantics of the UML models; thus we first sketch very roughly our setting. We assume that the semantics of a UML model is a set of UML-systems, precisely all those formally modelling the object communities described by the model itself plus additional information on those ob ject communities. In the following we will see examples of such extra information.

A *UML-system* is a labelled transition system $(U\_STATE, U\_LABEL, \rightarrow)$, where

- $U\_STATE \subseteq \wp(OBJECT\_STATE) \times \wp(LINK)$
- $LINK = ASSOCIATION \times O\_ID \times O\_ID$
- O_ID denotes the set of the ob ject identities
- $OBJECT\_STATE = O\_ID \times STATE$
- U_LABEL denotes the set of labels, each one representing a set of actions performed by the objects
- $\rightarrow \subseteq U\_STATE \times U\_LABEL \times U\_STATE$.

A state of a UML-system represents the states of the objects existing at a specific moment (identity, current values of attributes, actions that they are currently executing, and current control state, if their behaviour is defined by a state machine), and the existing links among them. A transition $us \xrightarrow{ul} us'$ represents the fact that some objects whose states are in us execute some actions (and as a possible consequence, exchange some messages, create and destroy some other objects) transforming it into us'. The label ul represents the performed actions.

A more detailed presentation of the UML-systems and of their use to model the UML semantics may be found in [14,15].

The class diagrams of a UML model determine a collections of UML-systems: those whose objects are defined by the classes in that class diagrams, and whose links are determined by the associations appearing in those class diagrams; obviously they satisfy all the constraints appearing in the class diagrams. Such collection is dened by the function $\mathsf{Sem}_{\mathsf{classDiagram}}$ that given a class diagram returns a subset of *UML_SYSTEM*.

The other diagrams (e.g., state machines, sequences, activity and collaboration diagrams) will either further restrict that collection of UML-systems, by dropping all those whose behaviour is not in agreement with them, or will provide some additional information. Formally, for each kind of diagrams, say diagramType, we have a function $\mathsf{Sem}_{\mathsf{diagramType}}$ associating with a diagram d of type diagramType an element in $(UML\_SYSTEM \rightarrow Bool) \times INFO$, where *INFO* is a set.

Summarizing, given a UML model *UM* defined as in Fig. 10 in the Appendix and denoting by Sem a semantics function for UML, we have:

$$\mathsf{Sem}(UM) \subseteq UML\_SYSTEM \times \ldots,$$

where *UML_SYSTEM* denotes the collection of all the possible UML-systems, and ... stands for the additional informations.

Moreover, if $UM = cd, d_1, \ldots, d_n$, where cd is a class diagram and $d_1, \ldots, d_n$ are diagrams of some different type

$$\mathsf{Sem}(UM) = (\{uSys \mid uSys \in USYS \wedge_{i=1,\ldots,n} restr_i(uSys)\}, \{info_1, \ldots, info_n\})$$

where

$$USYS = \mathsf{Sem}_{\mathsf{classDiagram}}(cd) \subseteq \mathcal{P}(UML\_SYSTEM),$$

and for $i = 1, \ldots, n$

$$(restr_i, info_i) = \mathsf{Sem}_{\mathsf{diagramType}_i}(d_i) \in (UML\_SYSTEM \rightarrow Bool) \times INFO_i.$$

We speak of a semantics function and not of the semantics function, since we can envisage various mode in which a diagram may contribute to the semantics of a UML model, as we will detail for the case of collaboration in the next section.

3.4 (Formal) Semantics of the UML Collaboration

The intuitive meaning of a collaboration, on the basis of the UML specification [3], can be formalized as follows. A collaboration determines the possible participants in a cooperation, by defining the roles that will play and which are their mutual relationships: i.e., they must be connected by some links (represented by the link connector) and must satisfy the condition expressed by the associated constraint.

We first consider the case of a collaboration with no associated behaviour diagrams. Thus our proposal for a formal meaning of a collaboration concerning its static aspects is the collection of all possible lists of participants.

Thus the initial proposal for a formal meaning of a collaboration concerning its structural aspects is the collection of all possible lists of participants.

Let *col* be a collaboration having the roles $R_1 : C_1, \ldots, R_n : C_n$.

$\mathsf{Stat\text{-}Sem}^{\mathsf{col}}(col) : U\_STATE \to (OBJECT\_STATE^{+} \to Bool)$

$\mathsf{Stat\text{-}Sem}^{\mathsf{col}}(col)(us)(os_1, \ldots, os_n)$ iff

– $os_1, \ldots, os_n$ are object states of us[6],
– $os_1 : C_1, \ldots, os_n : C_n$,
– for each link connector of *col* between R_i and R_j for the association ass there exists in us a link for the association ass between os_i and os_j, and
– if *col* has an associated constraint, such constraint holds when $R_1, \ldots, R_n$ are instantiated with $os_1, \ldots, os_n$ respectively.

This semantics of the collaboration will then be part of the additional information, describing which groups of ob jects may take part in that collaboration in each state.

If a collaboration has associated some behavioural diagrams we have to take them into account. We consider each different kind of behavioural diagram separately and discuss what they require, or better may require, when attached to a collaboration. Intuitively, it seems quite natural that the behavioural diagrams qualify the behaviour of the participants in the collaboration, but what that means precisely it is not so obvious, and there may be different ways to intend it.

To outline our viewpoints, we consider two cases of associated behavioural diagrams: sequence diagrams and state machine diagrams.

The sequence diagram case. If a collaboration has an associated sequence diagram, then the behaviour of the participants should be related to the traces defined by the sequence diagram itself. However, and this may be surprising, there is not a unique way to establish such relation. Indeed, we can give at least three ways to intend it, and we have examples of the use for each of them. At the notational level the sequence diagram attached to a collaboration should be tagged by a *mode* to make precise the way to intend it.

In the following we briefly present the three modes, but first we need a technical premise. In all three modes, and also later for the state machine case, we will make use of what we call semantics in isolation of a diagram, denoted by $\mathsf{Sem}^{\mathsf{is}}$. That semantic function captures the essential information provided by a diagram in isolation, i.e., without taking into account its model context, namely the other elements in the model. In general, given $\mathsf{Sem}^{\mathsf{is}}$, one can then proceed to give the semantics of the whole (or part of the) model that provides the context to that diagram.

Our simplified conceptual (meta)model of a sequence diagram is reported in Fig. 12 in the Appendix[7]; and here we briefly summarize our formal semantics of sequence diagrams according to [3, Sect. 14].

[6] Note that it is not required that all os_i be distinct.

[7] Here, for simplicity we do not consider the interaction combinators, nor the execution specifications.

Let *seq* be a sequence diagram as defined by the metamodel of Fig. 12.
$\mathsf{Sem}^{\mathsf{is}}(seq) \subseteq TRACE$[8]

- $TRACE = (LIFELINE \times OCCURRENCE)^*$
- *LIFELINE* denotes the set of the lifeline names
- $OCCURRENCE = SEND \cup REC \cup CREATE \cup DESTROY \cup INVARIANT$

If $\mathsf{Sem}^{\mathsf{is}}(seq) = traces$, then *traces* represent possible acceptable behaviours of the lifelines expressed in terms of sequences of (event) occurrences.[9]

Assume to have a collaboration *col* and an associated sequence diagram *seq* that are parts of a UML model *UM*, s.t. $\mathsf{Sem}^{\mathsf{is}}(seq) = traces$.

Mode 1. The sequence diagram *seq* attached to the collaboration *col* restricts the behaviour of the UML systems modelled by *UM*. This interpetation is useful, for example, when we use a collaboration for modelling a use case: the attached sequence diagram requires that the system and the actors must be able to perform the required scenarios.

$\mathsf{Behav\text{-}Sem}_1^{\mathsf{col}}(col, seq) : UML\_SYSTEM \to Bool$
$\mathsf{Behav\text{-}Sem}_1^{\mathsf{col}}(col, seq)(uSys) =$ for all *us* state of $uSys$,
if $\mathsf{Stat\text{-}Sem}^{\mathsf{col}}(col)(us)(os_1, \ldots, os_n)$, then
$Traces(us)|os_1, \ldots, os_n$ should include refinements of *traces*,

where $Traces(us)$ are all the possible traces starting from *us*, and $\_|os_1, \ldots, os_n$ restricts the traces to the occurrences happening on $os_1, \ldots, os_n$ (here for lack of room we do not define the refinement notion).

In this case the semantics of the collaboration will restrict the acceptable ob ject communities, i.e., the UML systems.

Mode 2. In some cases, the meaning of the collaboration behaviour is more descriptive than restrictive. Consider, e.g., the case of the description of an authentication procedure as a collaboration among a user, the protected resource and an authentication service. We do not want that each user in each moment should be forced to try to authenticate; we just want to define what is the behaviour following some procedure. Perhaps in the model there may be different collaborations with the same roles showing different ways to realize the authentication. The same case happens when we use the collaborations to model design patterns.

In this case the semantics of the collaboration, toghether the structural semantics, will include the traces defined by the sequence diagram.

$\mathsf{Behav\text{-}Sem}_2^{\mathsf{col}}(col, seq)$: $(U\_STATE \to (OBJECT\_STATE^+ \to Bool)) \times \mathcal{P}(TRACE)$

[8] Since the neg combinator is not part of the considered subset, we assume that a sequence diagram defines a set of acceptable traces, and we do not consider the unacceptable ones.

[9] Also the satisfaction of state invariants is included in the occurrences.

$\mathsf{Behav\text{-}Sem}_2^{\mathsf{col}}(col, seq) = (\mathsf{Stat\text{-}Sem}^{\mathsf{col}}(col), \mathsf{Sem}^{\mathsf{is}}(seq))$

This is an example of a semantics of a diagram resulting in some additional information (the set of traces).

Mode 3. We can also intend that the sequence diagram associated with a collaboration restricts the possible participants to those whose behaviour is in agreement with the traces defined by the latter. The restrictive modality is useful, for example, to describe groups of participants complying with a contract.

$\mathsf{Behav\text{-}Sem}_3^{\mathsf{col}}(col, seq)$: $U\_STATE \rightarrow (OBJECT\_STATE^{+} \rightarrow Bool)$

$\mathsf{Behav\text{-}Sem}_3^{\mathsf{col}}(col, seq)(s)(os_1, \ldots, os_n) =$

$\mathsf{Stat\text{-}Sem}^{\mathsf{col}}(col)(us)(os_1, \ldots, os_n)$ and $Traces(us)|_{os_1,\ldots,os_n}$ refines $\mathsf{Sem}^{\mathsf{is}}(seq)$, that is a set of traces.

The state machine diagram case. If the behavioural aspects of a collaboration are defined by a state machine associated with one of its roles, thus the behaviour of the participants playing that role should be related with the labelled transition system defined by the semantics in isolation of the state machine itself. However, as for the other kinds of behavioural diagrams, we cannot find a unique way to establish such relation, instead we can give at least three ways to intend it. As before for the sequence diagram case we first define the semantics of a state machine in isolation, and similarly to sequence diagram, the first semantics of the collaboration with the attached state machine will restrict the set of UML-systems, whereas the others will result in some additional information.

Our simplified conceptual (meta)model of the state machines is reported in Fig. 13 in the Appendix.

We briefly summarize here the formal semantics of state machine diagrams in isolation. Let sm be a state machine diagram as defined by the metamodel of Fig. 13 whose context is a class C.

$\mathsf{Sem}^{\mathsf{is}}(sm)$ is an *object-system*, i.e., a labelled transition system $(OBJECT\_STATE, OBJECT\_LABEL, \rightarrow^o, os_{init})$, where

- $OBJECT\_STATE$ is the collection of all the object states,
- $\rightarrow^o \subseteq OBJECT\_STATE \times OBJECT\_LABEL \times OBJECT\_STATE$.
- $OBJECT\_LABEL$ is the collection of the possible interactions of an object with the external world (receive/send an operation call, be destroyed/destroy another object, create another object).[10]
- $os_{init} \in OBJECT\_STATE$ is the initial state.

If $\mathsf{Sem}^{\mathsf{is}}(sm) = lts$, then the labelleld transition tree built by lts having os_{init} as initial state represents all possible lives of an object of class C.

[10] We assume that objects cannot directly update and read the attributes/associations of other objects.

Assume to have a collaboration *col* and a state machine diagram *sm* associated with the role $R : T$ s.t. $\mathsf{Sem}^{\mathsf{is}}(sm) = oSys$.

Mode 1. The state machine diagram *sm* attached to a collaboration *col* restricts the behaviour of the modelled UML systems; that is, whenever a group of objects are admissible participants of the collaboration, then the one playing the role R must behave accordingly to $oSys$. This interpretation is useful, for example, when we use a collaboration for modelling a use case: the state machine requires the system to behave in a given way whenever connected with proper actors.

$\mathsf{Behav\text{-}Sem}_1^{\mathsf{col}}(col, sm) : UML\_SYSTEM \to Bool$
$\mathsf{Behav\text{-}Sem}_1^{\mathsf{col}}(col, sm)(uSys) =$ for all us state of $uSys$,
if $\mathsf{Stat\text{-}Sem}^{\mathsf{col}}(col)(us)(os_1, \ldots, os_n)$ and os_i is playing the role R, then
$Tree(uSys, us, os_i)$ should refine $Tree(oSys, os_{init})$,

where $Tree(uSys, us, os_i)$ is the labelled transition tree having root os_i modelling its possible activities in the context of us as defined by $uSys$, and where $Tree(oSys, os_{init})$ is the transition tree having root os_{init} modelling its possible activities as defined by $oSys$ (here for lack of room we do not define the refinement notion).

Mode 2. In some cases, the meaning of the state machine behaviour is more descriptive than restrictive; it just describes a behaviour of a role.

In this case the semantics of the collaboration will include the transition tree defined by the state machine attached to the role R.

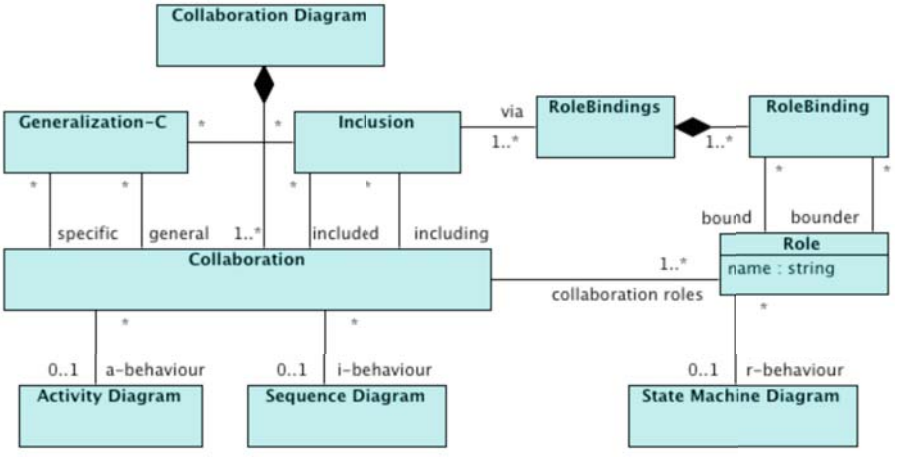
$\mathsf{Behav\text{-}Sem}_2^{\mathsf{col}}(col, sm) :$
$(U\_STATE \to (OBJECT\_STATE^+ \to Bool) \times$
$\quad LABELLED\_TRANSITION\_TREE)$
$\mathsf{Behav\text{-}Sem}_2^{\mathsf{col}}(col, sm) = (\mathsf{Stat\text{-}Sem}^{\mathsf{col}}(col), \mathsf{Sem}^{\mathsf{is}}(sm))$

Mode 3. We can also intend that the state machine diagram associated with the role R of a collaboration restricts the possible participants playing the role R to those whose behaviour is in agreement with the transition tree defined by the latter.

$\mathsf{Behav\text{-}Sem}_3^{\mathsf{col}}(col, sm) : U\_STATE \to (OBJECT\_STATE^+ \to Bool)$
$\mathsf{Behav\text{-}Sem}_3^{\mathsf{col}}(col, sm)(us)(os_1, \ldots, os_n) =$
$\mathsf{Stat\text{-}Sem}^{\mathsf{col}}(col)(us)(os_1, \ldots, os_n)$ and
$Tree(uSys, us, os_i)$ refines $Tree(oSys, os_{init})$.

3.5 Relationships between Collaborations

In the UML it is possible to establish some relationships among collaborations, first of all generalization/specialization (this is possible since collaboration is a specialization of classifier). In our conceptual (meta)model of the UML we prefer to define generalization/specialization specifically for the various constructs rather than to introduce a superclass of all the generalizable/specializable



Static Semantics
The general and the specific collaborations of a generalization must belong to the same collaboration to which the generalization belongs; similar constraint for inclusion
Let $<_G$ be the transitive closure of $<$ defined by $C_1 < C_2$ iff C_2 generalizes C_1, $<_G$ must be anti-reflexive and anti-symmetric
Let $<_I$ be the transitive closure of $<$ defined by $C_1 < C_2$ iff C_2 includes C_1,$<_I$ must be anti-reflexive and anti-symmetric
In an inclusion there should be a binding for each role of the included collaboration
A binding should bind a role of the including collaboration (binder role) with a role of the included collaboration (bound role)
The type of the binding role should be a subtype of the type of the bound role

Fig. 9. Collaboration relationships: conceptual (meta) model

constructs. This choice may not seem very elegant nor in the spirit of the OO approach, but it saves a lot of troubles, and we can skip:

– to find all the aspects of generalizations that are common to all the various cases (and to be sure to avoid mistakes when defining the special cases),

– to be careful not to put together the "being specializable/generalizable" with other aspects (in the UML metamodel classifier is the metaclass of the specializable elements, but it is also the metaclass of the instantiable elements, and thus collaborations may have instances),

– to give a meaning to this very abstract concept of specialization/generalization.

We think it is sensible to have a generalization/specialization notion for collaboration and we introduce it in our setting, see Fig. 9.

The UML 2 offers another way to establish a relationship between two collaborations: we may, e.g., define collaboration C_1 by means of a "use of collaboration C_2" (in this case the roles of C_2 will be bound to roles of C_1). It is useful also to employ the concept of collaboration use at the instance levels (i.e., in object diagrams) to present possible scenarios of objects taking part in various collaborations (in this case the roles of a collaboration will be bound to instance

specifications), see, e.g., Fig. 3. Thus we prefer to explicitly and separately define these two possible cases[11].

In Fig. 9 we present the conceptual (meta)model part concerning the relationships among collaborations. Now we try to clarify their semantics, but here, for lack of room, we detail only the inclusion relationship.

For what concerns the semantics at the static level, the participants in the including collaboration bound to the included one should be also possible participants in the last.

Assume that col_1 includes col_2 and that $os_1, \ldots, os_j$ are the participants of col_1 participating also in col_2, then the following condition must hold

$\mathsf{Sem}(col_1)(us)(os_1, \ldots, os_j, os_{j+1}, \ldots, os_n)$ implies $\mathsf{Sem}(col_2)(us)(os_1, \ldots, os_j)$.[12]

In the case of inclusion of collaboration with associated behaviour also the latter should be in agreement:

– if col_2 has associated a sequence diagram, then such sequence diagram should be called inside the sequence diagram associated with col_1 by means of the ref combinator;

– if an activity diagram is associated with col_2, then it should be called inside the associated activity diagram;

– if a role R_2 of col_2 bound with role R_1 of col_1 has associated a state machine SM_2, then also R_1 must have associated a state machine SM_1, and SM_1 should refine SM_2.

4 Conclusion

Motivated by the request coming from some companies with which we are engaged in a collaborative efgfort within a research laboratory, we have started to address on the basis of some real concrete cases the problem of moving from Business scenarios to SOA solutions. That has led us first to appreciate the need and value of modelling "cooperations", namely a community of objects cooperating to achieve a specific purpose, and then, when trying to adopt the UML collaboration that has been proposed for modelling such cooperations, we have experienced the difficulties and ambiguities linked with the use of that part of the UML.

The fact that collaborations play such an important role, as we have shown, in Business Modelling in general, and, more specifically, in the SoaML OMG proposal for a UML profile for SOA, gives a solid motivation for addressing that issue in the context of what we have called "well-founded methods" [6], of which our method MARS [12,13] is just an example.

Since UML is widely adopted and "understood", at least at the level of a visual support, instead of developing an entirely new concept, we have preferred to stick to the same visual notation, but redefining the syntax (metamodel) and

[11] For lack of room in this paper we do not cover the case of the instance level.

[12] [3, pag. 172] says that this condition is a semantic variation point.

analyzing its semantics in a critical way. Redefining a metamodel for development purposes is a suggestion of the inventors of the UML that we have found absolutely reasonable, because it eliminates the problem of checking the syntax against the official OMG specification of the UML. Indeed that specification has been organized in an interminable hierarchical splitting of concepts, dictated exclusively by the opportunity of permitting a flexible, highly reusable modularization of the supporting tools. The metamodel we propose, instead, allows us to make use of all concepts and model elements that are really useful, while it is not a real restriction, since it is well-known that a vast majority of the model element of the UML is practically of no use in the software development activity. Beside the syntactic aspects, we have addressed some basic semantic problems, especially related to the behavioural diagrams and thus to the behavioural aspects of collaboration. Rather surprisingly, different semantic modalities arise. In the "fresco" we are proposing, we have, admittedly, only sketched the basic possibilities, but enough, we believe, to convince that still a lot of problems have to be addressed, notwhitstanding the widespread use of the UML, with recurrent claims of a much improved semanitc clarification. That is exactly an area that we intend to explore further in rigour and detail, to fully achieve a well-founded approach.

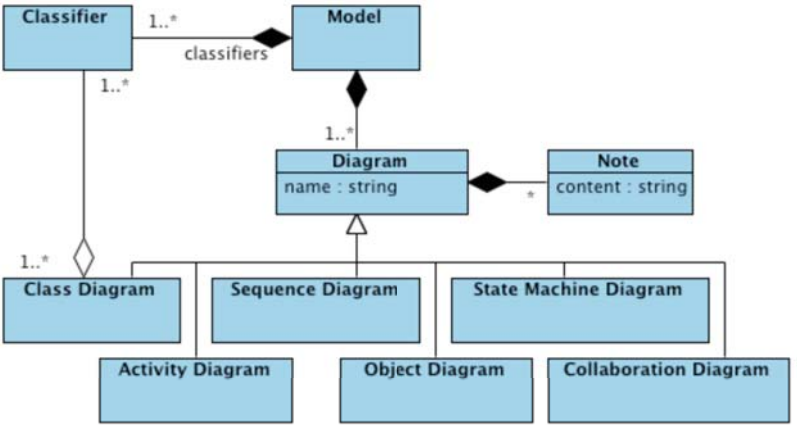
Acknowledgements. We gratefully acknowledge the help provided by the very accurate and stimulating reviews.

References

1. OMG: Service oriented architecture Modeling Language (SoaML) - Specification for the UML Profile and Metamodel for Services, UPMS (2008)
2. Astesiano, E., Reggio, G., Ricca, F.: Modeling Business within a UML-Based Rigorous Software Development Approach. In: Degano, P., De Nicola, R., Meseguer, J. (eds.) Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 261–277. Springer, Heidelberg (2008)
3. UML Revision Task Force: OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2 (2007)
4. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual, 2nd edn. The Addison-Wesley Object Technology Series. Addison-Wesley, Reading (2004)
5. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide, 2nd edn. The Addison-Wesley Object Technology Series. Addison-Wesley, Reading (2005)
6. Astesiano, E., Reggio, G., Cerioli, M.: From Formal Techniques to Well-Founded Software Development Methods. In: Aichernig, B.K., Maibaum, T. (eds.) Formal Methods at the Crossroads. From Panacea to Foundational Support. LNCS, vol. 2757, pp. 132–150. Springer, Heidelberg (2003)
7. Meyer, B.: Object-Oriented Software Construction. Prentice-Hall, Englewood Cliffs (1997)
8. Jones, C.: Some Mistakes I Have Made and What I Have Learned FromThem. In: Astesiano, E. (ed.) ETAPS 1998 and FASE 1998. LNCS, vol. 1382, p. 7. Springer, Heidelberg (1998)

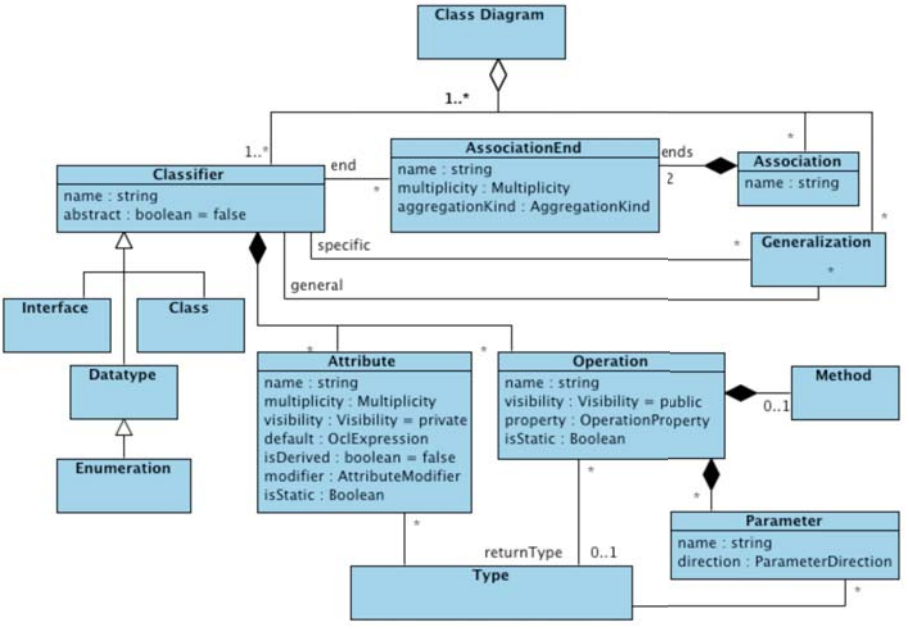
9. Engels, G., Hausmann, J., Heckel, R., Sauer, S.: Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In: Evans, A., Kent, S., Selic, B. (eds.) UML 2000. LNCS, vol. 1939, pp. 323–337. Springer, Heidelberg (2000)
10. Broy, M., Crane, M.L., Dingel, J., Hartman, A., Rumpe, B., Selic, B.: 2^{nd} UML 2 semantics symposium: Formal semantics for UML. In: Kühne, T. (ed.) MoDELS 2006. LNCS, vol. 4364, pp. 318–323. Springer, Heidelberg (2007)
11. Cengarle, M., Knapp, A., Tarlecki, A., Wirsing, M.: A heterogeneous approach to UML semantics. In: Degano, P., De Nicola, R., Meseguer, J. (eds.) Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 383–402. Springer, Heidelberg (2008)
12. Astesiano, E., Reggio, G.: Towards a well-founded UML-based development method. In: SEFM 2003. ACM Press, New York (2003)
13. Astesiano, E., Reggio, G.: Tight Structuring for Precise UML-based Requirement Specifications. In: Wirsing, M., Knapp, A., Balsamo, S. (eds.) RISSEF 2002. LNCS, vol. 2941, pp. 16–34. Springer, Heidelberg (2004)
14. Reggio, G., Astesiano, E., Choppy, C., Hussmann, H.: Analysing UML Active Classes and Associated State Machines – A Lightweight Formal Approach. In: Maibaum, T. (ed.) FASE 2000. LNCS, vol. 1783, p. 127. Springer, Heidelberg (2000)
15. Reggio, G., Cerioli, M., Astesiano, E.: Towards a Rigorous Semantics of UML Supporting its Multiview Approach. In: Hussmann, H. (ed.) FASE 2001. LNCS, vol. 2029, p. 171. Springer, Heidelberg (2001)

A The Conceptual (Meta)Model of the UML



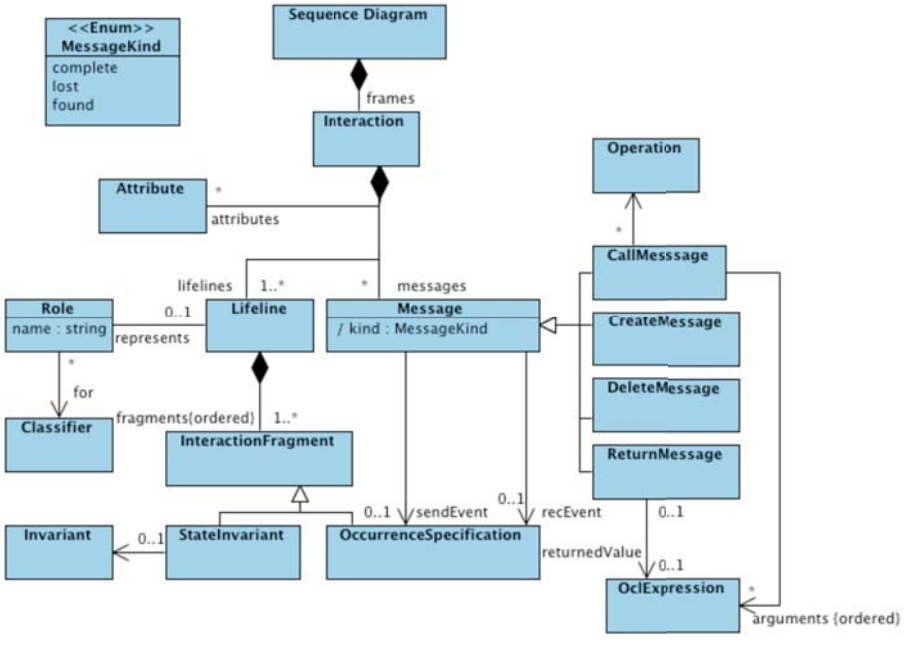
Static Semantics
M.classifiers = M.diagram->select(oclIsTypeOf(Class Diagram)).classifier (that implicitly requires that each model must include at least one class diagram)
All classes in a model must have distinct names

Fig. 10. UML model structure: conceptual (meta) model



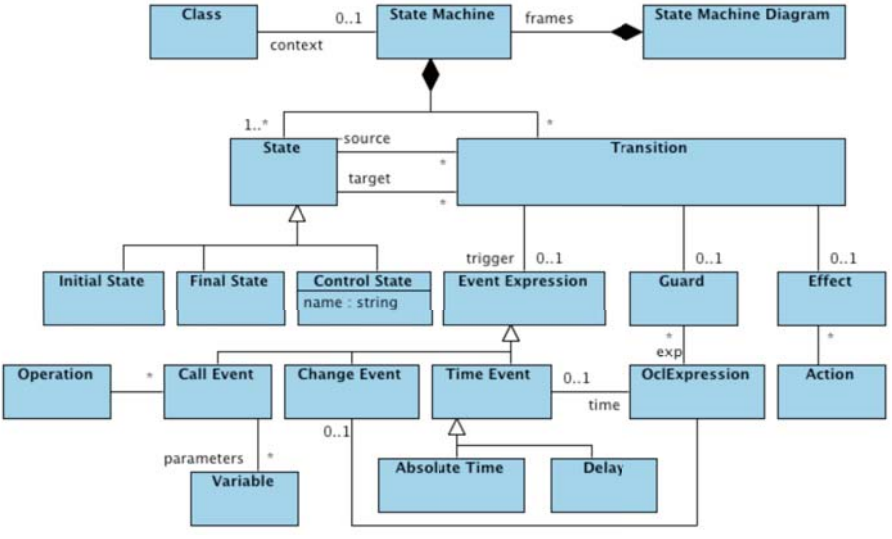
Static Semantics
An interface is abstract and has no attributes
All attributes must have distinct names, and all operations with the parameters having the same types must have distinct names
The transitive closure of the generalization relationship among classes must be without cycles

Fig. 11. Class diagram: conceptual (meta) model



The type of a role must be either a class or an interface
If the kind of a message is complete, then both the send and the rec events are present; if lost, then only the send event is present; otherwise, namely if lost, only the rec event is present
The events of a message are on a lifeline of the interaction and vice versa
The operation of a message must belong to the class/interface of the receiving lifeline
The types of the arguments of a message are in agreement with those of the corresponding operation
The arguments and the returned values of a message must be correct in the environment containing self typed as the role of the sending lifeline and the interaction attributes

Fig. 12. Sequence diagram: conceptual (meta)model



Static Semantics
A state machine must have one and only one initial state
The context of a state machine must be a class of the model
The source of a transition cannot be a final state
The target of a transition cannot be an initial state
If a transition has no trigger, then its source is an initial state
The expression building a change-event of a transition is wff w.r.t. the environment made by the self typed as the context, and has type Boolean
The expression building a timed-event of a transition is wff w.r.t. the environment made by the self typed as the context, and has type Time
The guard of a transition is wff w.r.t. the environment made by the self typed as the context and the parameters of the event of the same transition, and has type Boolean
The effect of a transition is wff w.r.t. the environment made by the self typed as the context and the parameters of the event of the same transition
The event parameters and the parameters of the operation corespond in number and type

Fig. 13. State machine: conceptual (meta) model

Client Synthesis for Aspect Oriented Web Services

Mehdi Ben Hmida[1] and Serge Haddad[2]

[1] LAMSADE, CNRS & Université Paris-Dauphine, France
mehdi.benhmida@lamsade.dauphine.fr
[2] LSV, CNRS & ENS Cachan, France
serge.haddad@lsv.ens-cachan.fr

Abstract. Client synthesis for complex Web services is a critical and still open topic as it will enable more flexibility in the deployment of such services. In previous works, our team has developed a theoretical framework based on process algebra that has led to algorithms and tools for the client interaction. Here, we show how to generalise our approach for aspect oriented Web services.

1 Introduction

From elementary Web services to complex ones. Web services are self contained, self-describing modular applications that can be published, located, and invoked across the Web. They are based on a set of independent open platform standards to reach a high level of acceptance. Web services framework is divided into three areas: communication protocol, service discovery and service description. The "Web Services Description Language" (WSDL) [26] provides a formal, computer-readable description of Web services. Such a description specifies the software component interfaces listing the collection of operations that are network accessible through standard XML messaging. It includes all information that an application needs to invoke such as the message structure, the response structure and some binding information like the transport protocol, the port address, etc.

However simple operation invocation is not sufficient for some kind of composite services. They require in addition a long-running interaction derived by an explicit process model. This kind of services may often be encountered in two cases. First when a Web service is developed as an agent, it is composed by a set of accessible operations and a process model which schedules the invocation to a correct use of the service. Secondly, facing to the capability limits of Web services, composite services may be obtained by aggregating existing Web services in order to create more sophisticated services (and this in a recursive way).

In order to deal with the behavioural aspects of complex services, some industrial and academic specifications languages have been introduced. Among them, "Business Process Execution Language for Web Services" (BPEL4WS or more succinctly BPEL) has been proposed by leading actors of industry (BEA, IBM, and Microsoft) and has quickly become a standard [14].

The two facets of complex Web services. BPEL supports two different types of business processes (see for instance [16], [17]):

C. Choppy and O. Sokolsky (Eds.): Monterey Workshop 2008, LNCS 6028, pp. 24–42, 2010.

- Executable processes specify the exact details of business processes. They can be executed by an orchestration engine.
- Abstract business protocols specify the public message exchange between the client and the service. They do not include the internal details of process flows but are required in order for the client to correctly interact with the service.

Given the description of an executable process, its associated interaction protocol is obtained by an abstraction mechanism (which masks all the internal operations of the service). However the issues raised by these two types of processes are very different. A specification of an executable process is close to the definition of a program whereas the specification of interaction protocol mainly raises an difficult problem: how to synthetize a client which will correctly handle the interaction with the service.

The synthesis problem. Indeed by construction, the external behaviour of a service is non deterministic due to its internal choices. It is then *a priori* unclear whether a client, i.e. a deterministic program, can be designed to interact with it. Furthermore the specification often includes timing constraints (e.g. implicit detection of the withdrawal of an interaction by the client) implying that these timing constraints must also be taken into account by the client. However since no semantics of the interaction process is given for BPEL (not to be confused with the semantics of the service execution), this problem could not be formally stated. In practice, the industrial products including predefined clients assume a simple interaction protocol as proposed by WSDL (like for instance a "query-answer" interaction). Thus it is clear that the synthesis problem is a critical issue for the rapid deployment of composite services.

Adaptation and Web services. Aspect oriented programming (AOP) helps the programmer to isolate non functional software (like authentication and logging) from business software. Using AOP eases the modification of implemented policies as it does not impact the functional part. However in the context of JAVA, it requires either to change the compiler, the loader or the virtual machine. AOP is also desirable for Web services since they require a lot of non functional codes but the integration of AOP in a Web service framework raises significant difficulties.

Previous contributions. In our previous works, we have addressed both service adaptability and client interaction issues but separately.

First, we have specified what is an external behaviour, i.e. we have given an operational semantics to an abstract BPEL specification in terms of a time transition system [10,11]. The semantics is obtained by a set of rules in a modular way. Given a constructor of the language and the behaviour of some components, a rule specifies a possible transition of a service built via this constructor applied on these components. As previously discussed, the transition system is generally non deterministic. Then we have defined a relation between two communicating systems which formalizes the concept of a correct interaction. There are standard relations between dynamic systems like the language equivalence and the bisimulation equivalence but none of them matches our needs. Thus we have introduced the interaction relation which can be viewed as a bisimulation relation modified in order to capture the nature of the events (i.e. the sending of a message is an action whereas the reception is a reaction). Afterwards we have focused on the synthesis of a client which is in an interaction relation with the transition

system corresponding to the system. The client we look for must be implementable, in other words it should be a deterministic automaton. It has appeared that some BPEL specifications do not admit such a client i.e. they are inherently ambiguous. Thus the algorithm we have developed either detects the ambiguity of the Web service or generates a deterministic automaton satisfying the interaction relation. The core of this algorithm is a kind of determinisation of the transition system of the service.

Independently we have proposed an Aspect Oriented Programming (AOP) [18] approach which aims to change elementary Web services at runtime [2,23].

Our contributions. Here, we extend these works by providing:

- A method to design, deploy and publish *aspect-oriented and composite* Web services;
- A formal semantics for aspect-oriented Web services;
- An algorithm that generates a client (i.e. an automata) based on this semantics or detect that the service is ambiguous. Observe that this generation takes into account specification of aspects with the aim to dynamically create additional automata at run time.
- A client interpreter able to handle interactions not fully specified in the published Web service description. In particular, it simultaneously manages execution of several automata with synchronization contrary to our previous approach that manages a single automaton.

This paper is organized as follows. Section 2 details the approach for synthesis of client for service without adaptation. Section 3 presents the generalisation to aspect oriented web services. Section 4 discusses related work. Finally in section 5 we conclude and give some perspectives to this work.

2 Client Synthesis for Web Services

In this section, we develop the principle of client synthesis for services *without* adaptation.

2.1 A Formal Semantics for BPEL Abstract Processes

BPEL provides a set of operators describing in a modular way the observable behaviour of an abstract process. As shown in [22], this kind of process description is close to the process algebra paradigm illustrated for instance by CCS [20], CSP [13] and ACP [3]. However, time is explicitly present in some of the BPEL constructors and thus the standard process algebra semantics are inappropriate for the semantics of such a process. In order to model time, we have chosen a discrete time semantics since on the one hand the theory of dense time is more involved and on the other hand timing requirements in Web services are simpler than in real-time systems. For sake of clarity, we have not formalized more technical features of BPEL like the compensation handlers. Our semantics associates a finite automaton with an abstract process.

The alphabet of the automaton. The first step for the definition of a semantics consists in specifying the action alphabet for a BPEL process. We have five kinds of actions:

- A time unit elapsing is denoted by χ.
- Silent actions, denoted by τ cannot be observed by the client. They correspond to decisions taken by the server (evaluation of a condition for switch, while, etc.).
- Exceptions; the set of exception events is denoted by Ex.
- In order to control that the client correctly detects the end of the service, we introduce $\surd$, the termination event. This action will also simplify the definition of the operational semantics.
- Sending and receiving messages: the set of types of messages will be denoted by M. The emission is denoted by $!m$ and the reception is denoted by $?m$. We also set $!M = \{!m \,|\, m \in M\}$ and $?M = \{?m \,|\, m \in M\}$ and the wildcard $*$ may be substituted for ! or ?.

Actions different from time elapsing can be classified as immediate (τ, $\surd$ and exceptions) or delayed (emissions and receptions). The first kind of actions are performed in null time (w.r.t. the time scale) and thus in our semantics have priority over the other actions including time elapsing.

The states of the automaton. Each state will be associated with a BPEL process obtained by successive transformations from the initial process. Two states have different associated processes. At the beginning of the construction, there is a single state (the initial one) corresponding to this process. Each time an edge is defined, a new process is computed and if this process does not label an existing state then such a state is created. Due to the semantic rules given in the next subsection, it can be proved that the number of derived processes is finite (and thus the number of states is also finite).

The transitions of the automaton. The transitions starting from a state are obtained by a top-down analysis of the process expression labelling this state. This analysis is usually defined with the help of operational semantic rules. The definition of a semantic rule $[op_x]$ for a generic process $P = op_x(P_1, P_2, \ldots)$ includes the following parts:

- a boolean expression over some potential transitions of selected components of P: $Bexp(\{P_{o(i)} \xrightarrow{\alpha_i} P'_{o(i)}\})$;
- this condition is enforced by a second condition on the occurring labels denoted by $guard(\{\alpha_i\})$.
- If the two conditions are fulfilled then a state transition for P is possible where the label $Lexp(\{\alpha_i\})$ is an expression depending on the labels of subprocesses transition and
- the new process is an expression $Nexp(P, \{P'_{o(i)}\})$ depending on the original process and the new subprocesses.

So, a generic rule, presented with the usual style has the following structure:

$$[op_x] : \frac{Bexp(\{P_{o(i)} \xrightarrow{\alpha_i} P'_{o(i)}\})}{P \xrightarrow{Lexp(\{\alpha_i\})} Nexp(P, \{P'_{o(i)}\})} \text{ where } guard(\{\alpha_i\})$$

For sake of readability, we do not follow the (verbose) XML syntax of a BPEL process. Instead we have chosen a simplified syntax close to the one used for process algebra whose meaning should be immediate for who knows BPEL. As usual, we begin the definition of rules by giving the ones corresponding to the basic processes of BPEL. These basic processes are `empty`, $?o[m]$, $!o[m]$ and `throw`$[e]$.

The `empty` *process* `empty` can only terminate (the notation 0 is the null process).

$$\texttt{empty} \xrightarrow{\surd} 0$$

The $?o[m]$ *and* $!o[m]$ *processes.* The process $?o[m]$ (which corresponds to the input operation of WSDL) consists in receiving a message of type m. The process $!o[m]$ (which corresponds to the notification operation of WSDL) consists in sending a message of type m. We only consider these two types of WSDL operations. The two other types can be built with the sequence constructor (see below). Since these actions are not immediate, time can elapse. This leads to the two rules below.

$$*o[m] \xrightarrow{\chi} *o[m] \qquad *\, o[m] \xrightarrow{*m} \texttt{empty} \quad \text{with} * \in \{?, !\}$$

The `throw` *process.* The process `throw`$[e]$ raises an exception e which must be catched in some `scope` process.

$$\texttt{throw}[e] \xrightarrow{e} 0$$

We also introduce an auxiliary process `time` that represents time elapsing (not present in the BPEL definition).

$$\texttt{time} \xrightarrow{\chi} \texttt{time}$$

The `sequence` *process (;).* The process $P\ ;\ Q$ executes the process P then the process Q. Since the operator ";" is associative, we safely restrict the number of operands to two processes. The `sequence` process acts as its first subprocess while this process does not indicate its termination. In the latter case, the `sequence` process acts as the second process can do.

$$\frac{P \xrightarrow{a} P'}{P\ ;\ Q \xrightarrow{a} P'\ ;\ Q} \text{ where } a \neq \surd$$

$$\frac{P \xrightarrow{\surd} \text{ and } Q \xrightarrow{a} Q'}{P\ ;\ Q \xrightarrow{a} Q'}$$

Remark. The set of rules will imply that if there is an action $a \neq \surd$ such that $P \xrightarrow{a} P'$, then $P \xrightarrow{\surd}$ cannot occur.

The `switch` *process.* The process `switch`$[\{P_i\}_{i\in I}]$ chooses to behave as one process among the set $\{P_i\}$. Each branch of its execution is guarded by an *internal* condition. Conditions are evaluated w.r.t. the order of their appearance in the description. However since the client has no way to predict the choice of the service, this order is irrelevant. The main consequence is that from the point of view of the client, *this choice is non deterministic*. The `switch` process becomes one of its subprocesses in a silent way. Let

us note that we have implicitly supposed that at least one condition is fulfilled. In the other case, it is enough to add the process `empty` as one of the subprocesses.

$$\forall\, i \in I \ \texttt{switch}[\{P_i\}_{i\in I}] \xrightarrow{\tau} P_i$$

The `while` *process.* The process $\texttt{while}[P]$ iterates an inner process as long as an *internal* condition is satisfied. Like `switch`, `while` evaluates in a silent way its condition. Thus we have two rules depending on this internal evaluation.

$$\texttt{while}[P] \xrightarrow{\tau} P \ ;\ \texttt{while}[P]$$

$$\texttt{while}[P] \xrightarrow{\tau} \texttt{empty}$$

The `flow` *process.* The process $\texttt{flow}[\{P_i\}_{i\in I}]$ simultaneously activates a set of processes $\{P_i\}$. For the moment considering that the synchronization primitives of BPEL are internal ones we have not yet implemented this synchronization. Thus this parallel execution is similar to a "fork-join" in the sense that the combined process ends its interaction when all subprocesses have completed their execution. Subprocesses of a `flow` process act independently except for one action: they simultaneously indicate their termination. In the latter case, the `flow` process becomes the null process. Furthermore internal actions are considered as immediate and consequently the occurrence of such an action in a subprocess prevents the occurrence of a delayed action (sending or reception of a message) in another subprocess.

• Individual actions:

1.

$$\forall j \in I \ \frac{P_j \xrightarrow{a} P'}{\texttt{flow}[\{P_i\}_{i\in I}] \xrightarrow{a} \texttt{flow}[\{P_i\}_{i\in I\setminus\{j\}} \cup \{P'\}]} \text{ where } a \in Ex \cup \{\tau\}$$

2.

$$\forall\, m \in M\ \forall j \in I \ \frac{P_j \xrightarrow{*m} P' \wedge \forall\, i \in I\ \forall\, a \in Ex \cup \{\tau\},\ \neg P_i \xrightarrow{a}}{\texttt{flow}[\{P_i\}_{i\in I}] \xrightarrow{*m} \texttt{flow}[\{P_i\}_{i\in I\setminus\{j\}} \cup \{P'\}]}$$

• Time elapsing: all processes must either let time elapse or terminate.

$$\forall J \neq \emptyset\ J \subseteq I \frac{\forall i \in J\ P_i \xrightarrow{\chi} P_i' \wedge \forall i \in I \setminus J\ P_i \xrightarrow{\surd}}{\texttt{flow}[\{P_i\}_{i\in I}] \xrightarrow{\chi} \texttt{flow}[\{P_i'\}_{i\in J} \cup \{P_i\}_{i\in I\setminus J}]}$$

• Termination:

$$\frac{\forall\, i \in I\ P_i \xrightarrow{\surd}}{\texttt{flow}[\{P_i\}_{i\in I}] \xrightarrow{\surd} 0}$$

The `scope` *process* $\texttt{scope}(P, E^d)$ with

$$E^d \stackrel{def}{=} [\{(m_i, P_i) \,|\, i \in I\}, (d, Q), \{(e_j, R_j) \,|\, j \in J\}]$$

may evolve due to P evolution, reception of a message m_i, expiration of the timeout with duration d or occurrence of an exception e_j. We note $M_I = \{m_i \,|\, i \in I\}$ and $E_J = \{e_j \,|\, j \in J\}$.

• P actions: The termination exits the scope whereas another action does not.

$$\frac{P \xrightarrow{\surd}}{\texttt{scope}(P, E^d) \xrightarrow{\surd} 0}$$

$$\frac{P \xrightarrow{a} P'}{\texttt{scope}(P, E^d) \xrightarrow{a} \texttt{scope}(P', E^d)} \text{ where } a \notin Ex \cup M_I \cup \{\surd, \chi\}$$

• Receiving a message m_i:

$$\forall\, i \in I \;\; \frac{\forall\, a \in Ex \cup \{\tau, \surd\},\ \neg P \xrightarrow{a}}{\texttt{scope}(P, E^d) \xrightarrow{?m_i} P_i}$$

• Exception handling: which depends whether the raised exception is catched in this scope.

$$\forall\, j \in J \;\; \frac{P \xrightarrow{e_j}}{\texttt{scope}(P, E^d) \xrightarrow{\tau} R_j}$$

$$\forall\, e \in Ex \setminus E_J \;\; \frac{P \xrightarrow{e}}{\texttt{scope}(P, E^d) \xrightarrow{e} 0}$$

If an exception e is never catched at any level then the process is an erroneous one which can straightforwardly checked by examining whether an exception labels an transition of the automaton.

• Time elapsing

$$\forall\, d > 0 \;\; \frac{P \xrightarrow{\chi} P'}{\texttt{scope}(P, E^d) \xrightarrow{\chi} \texttt{scope}(P', E^{d-1})}$$

• Time out

$$\frac{P \xrightarrow{\chi}}{\texttt{scope}(P, E^1) \xrightarrow{\chi} Q}$$

The `pick` *process* can be viewed as a particular case of the `scope` process:

$$\texttt{pick}(E^d) \equiv \texttt{scope}(\texttt{time}, E^d)$$

2.2 Interaction Relation

We first informally state what should be a correct interaction between two automata. As for the bisimulation relation, we require a relation between pairs of states of the two systems. Obviously the pair consisting of the initial states should belong to this relation.

Furthermore, the states of a pair should have a coherent view of the next interaction steps to occur. At first, this implies that the relation must take into account the mutually observable steps. Thus we introduce the observable transition relation of an automaton by $s \stackrel{a}{\Rightarrow} s'$ iff $s \stackrel{\tau^* a \tau^*}{\rightarrow} s'$, $s \stackrel{\epsilon}{\Rightarrow} s'$ iff $s \stackrel{\tau^*}{\rightarrow} s'$.

Once it is done, we could require (like for bisimulation) that if a state s of the pair (s, s') may evolve by an observable transition of its automaton to some new state s_1, s' should have a similar observable transition leading to a state s'_1 which would compose with s_1, a new pair of consistent views.

However we need to be careful. First, if an automaton sends a message the other one must be able to receive the message. So it is necessary to introduce the notion of complementary actions $\overline{?m} = !m$, $\overline{!m} = ?m$ and $\forall a \notin \{!m\}_{m \in M} \cup \{?m\}_{m \in M}\ \overline{a} = a$ and to require that the synchronized evolution is obtained via complementary actions.

But this requirement is too strong as it does not capture the different nature of the sending and reception of a message. A sending is an action whereas a reception is a reaction and will not spontaneously occur. Therefore a more appropriate relation will first require that if, in s belonging to the pair (s, s'), an automaton may receive a message m, then there is a third state s'' of the other automaton indistinguishable from s' w.r.t. the observable transitions which can send m and second that in s' the other automaton can send a message (not necessarily m). The first condition expresses that the former automaton is not overspecified and the second one that it will not wait indefinitely for a message.

These considerations yield the following formal definition.

Definition 1 (Interaction relation). *Let $A_1 = (S, s_{01}, A, \rightarrow_1)$ and $A_2 = (S, s_{02}, A, \rightarrow_2)$ be two automata. Then A_1 and A_2* correctly interact *iff $\exists \sim \subseteq S_1 \times S_2$ such that:*

- $s_{01} \sim s_{02}$
- $\forall s_1, s_2$ *such that* $s_1 \sim s_2$
  - *Let $a \notin \{?m \mid m \in M\}$ then*
    - *if* $\exists\, s_1 \stackrel{a}{\Longrightarrow}_1 s'_1$, *then* $\exists\, s_2 \stackrel{\overline{a}}{\Longrightarrow}_2 s'_2$ *with* $s'_1 \sim s'_2$
    - *if* $\exists\, s_2 \stackrel{a}{\Longrightarrow}_2 s'_2$ *then* $\exists\, s_1 \stackrel{\overline{a}}{\Longrightarrow}_1 s'_1$ *with* $s'_1 \sim s'_2$
  - *Let $m \in M$; if* $s_1 \stackrel{?m}{\Longrightarrow}_1 s'_1$ *then*
    - $\exists\, s_2^- \stackrel{w}{\Longrightarrow}_2 s_2$, $\exists\, s_2^- \stackrel{w}{\Longrightarrow}_2 s_2^+$, $\exists\, s_2^+ \stackrel{!m}{\Longrightarrow}_2 s'_2$ *with* $s_1 \sim s_2^+$ *and* $s'_1 \sim s'_2$ *where w is a word*
    - $\exists\, s_2 \stackrel{!m'}{\Longrightarrow}_2 s'_2$
  - *Let $m \in M$; if* $s_2 \stackrel{?m}{\Longrightarrow}_2 s'_2$ *then*
    - $\exists\, s_1^- \stackrel{w}{\Longrightarrow}_1 s_1$, $\exists\, s_1^- \stackrel{w}{\Longrightarrow}_1 s_1^+$, $\exists\, s_1^+ \stackrel{!m}{\Longrightarrow}_1 s'_1$ *with* $s_1^+ \sim s_2$ *and* $s'_1 \sim s'_2$ *where w is a word*
    - $\exists\, s_1 \stackrel{!m'}{\Longrightarrow}_1 s'_1$

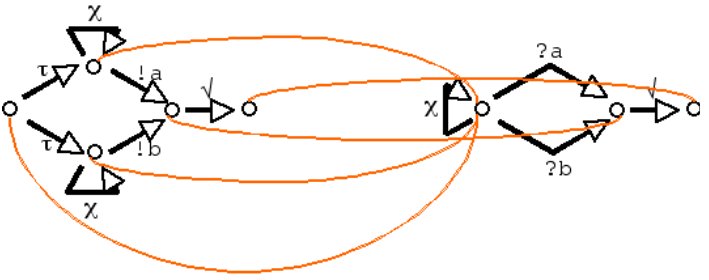


Fig. 1. An example of two interacting systems

We illustrate the interaction relation in figure 1. The left automaton corresponds to the BPEL service $switch[!o[a], !o[b]]$, i.e. a service that internally decides to send message a or b. The right automaton represents a possible client that waits either for an a or b before terminating. The curved lines denote the relation between states. Observe the importance of the definition: the initial state of the left automaton is related with three states, on that can (observably) send both a and b and the two other ones that can only send either a or b. However for the client these states are not "distinguishable" and so this initial state is allowed to wait for an a or a b.

2.3 Client Automaton Synthesis

We are now in position to present the client synthesis algorithm. Since the client must be implementable, we require it to be *deterministic*. This consideration leads to choose as model for our client a deterministic automaton which is in interaction relation with the automaton of the BPEL process.

Before developing it, we emphasize that there exist BPEL process which do not admit clients. For instance, process $\texttt{switch}[?o[m], ?o[m']]$ internally chooses to receive either a message m or m' and thus no deterministic automaton can correctly interact with it since it would imply that, in its initial state, the client should send either m or m' while the server would wait the other message. Observe the difference with process $\texttt{switch}[!o[m], !o[m']]$ where a client can be easily designed: it just waits for either m or m' (see figure 1). We say that a process is *ambiguous* if it does not admit a deterministic automaton which is in interaction relation with it.

Here we give an abstract view of the algorithm. A detailed description of the algorithm is given in [10]. The general principle of our algorithm is similar to a determinisation procedure: a state of the TA client will correspond to a subset of states of the TA of the service.

More precisely, each potential state s of the automaton client is associated with a subset of states $S_2(s)$ of the TA service which are related to s via the interaction relation. During the construction, there is a stack of client states to be processed. At the beginning of the algorithm, the stack contains an initial client state s_{01} such that $S(s_{01}) = \{s_{02}\}$, s_{02} being the initial state of the service. It stops either when the stack is empty (i.e. the client has been built) or when it has detected the ambiguity of the service.

First, we compute the ϵ-closure by τ-transitions. If this subset (call it S') of service states is already associated with a state s' of the client, then the transition of the client

which has generated the subset is redirected to s'. Otherwise, one creates a new client state and we go on. We check the interaction relation for transitions. If it is not fulfilled then we stop the construction. We give below an algorithmic description of a step of the algorithm.

unstack (s, S')
$S' \leftarrow \epsilon - closure(S')$
If S' has already be analysed and paired with s' **Then**
one redirects the arc entering s toward s' and one deletes s
Else
For every a s.t. subset S_a of a-successors of S' is non empty **do**
If $a \notin \{!m\}_{m \in M}$ and $\exists t \in S'\ \neg t \stackrel{a}{\Longrightarrow}$ **Then** ambiguity
Else If $a \in \{!m\}_{m \in M}$ and $\exists t \in S'\ \forall m'\ \neg t \stackrel{!m'}{\Longrightarrow}$ **Then** ambiguity
Else create s_a; add $s \stackrel{a}{\longrightarrow} s_a$; stack (s_a, S_a)

2.4 Client Interpreter

We have implemented a client interpreter based on the previous theoretical developments. The interpreter downloads the BPEL description of the service. Then it generates the automaton according to the algorithm and it "executes" this automaton (see figure 2). More precisely:

- It maintains the current state of the automaton.
- It opens (or let open) one input window per enabled action $!m$; this means that the user can choose the type of message it wants to send and to enter the corresponding data. It closes the windows that correspond to messages now disabled.
- It arms a time-out of one time unit.
- It changes the current state,
  - either on reception of a message with opening an output window,
  - either on validation of an input window with sending of the message,
  - or on triggering of time-out.

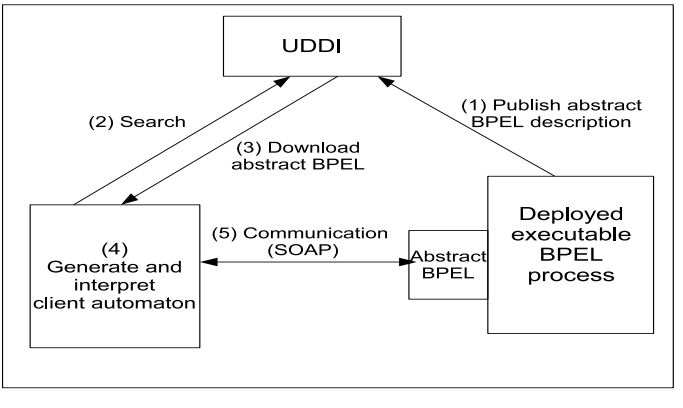


Fig. 2. Generic client interpreter

3 Aspect Oriented Programming and Web Services

3.1 Principles of AOP

AOP is a concept that enables the modularization of crosscutting concerns into single units called aspects, which are modular units of crosscutting implementation [18]. Crosscutting concerns are requirements that cannot be localized to an individual software component and that impact many components. In aspect-speak, these requirements cut across several components. Aspect-oriented languages [19,15,1,21] are based on three paradigms:

1. *Joinpoints*: They denote the locations in the program that are affected by a particular crosscutting concern.
2. *Pointcuts*: They specify a collection of conditional joinpoints.
3. *Advices*: They are codes that are executed *before*, *after* or *around* a joinpoint.

For instance the logging functionality is often scattered horizontally across object hierarchies and has nothing to do with the core functions of the objects it is scattered across. The same is true for other types of code, such as security, exception handling, and transparent persistency. This scattered and unrelated code is known as crosscutting code and is the reason for AOP's existence. Using AOP, we can insert the logging code into the classes that need it with a tool called a *weaver*. This way, objects can focus on their core responsibilities. The figure 3 shows the weaving process. The weaver is in charge for taking the code specified in a traditional (base) programming language, and the additional code specified in an aspect language, and merging the two together. The weaver has the task to process aspects and component code in order to generate the specified behaviour. The weaver inserts the aspects in the specified joinpoint transversally. The weaving can occur at compile time (modifying the compiler), load time (modifying the class loader) or runtime (modifying the interpreter).

3.2 Adapting BPEL Processes

Several researches [4,5,25] consider AOP as an answer to improve WS flexibility. In our previous approach, we developed an AOP-based tool named *Aspect Service Weaver* (*ASW*) [2,23]. The ASW intercepts the SOAP messages between a client and an elementary WS, then it verifies during the interaction whether there is a newly introduced behaviour (*advice service*). We use the AOP weaving time to add the new behaviour (*before*, *around* or *after* an activity execution). The advices services are elementary WSs whose references are registered in a file called "*aspect services file descriptor*". The pointcut language is based on XPath [27]. XPath queries are applied on the service description (WSDL) to select the set of methods on which the advice services are inserted.

We extend this approach to BPEL processes. We apply the AOP concepts to a BPEL process in order to automatically generate an extended BPEL process without modifying the BPEL engine. This process contains the base BPEL process and the advice services. We apply the AOP concepts on BPEL processes in the following way:

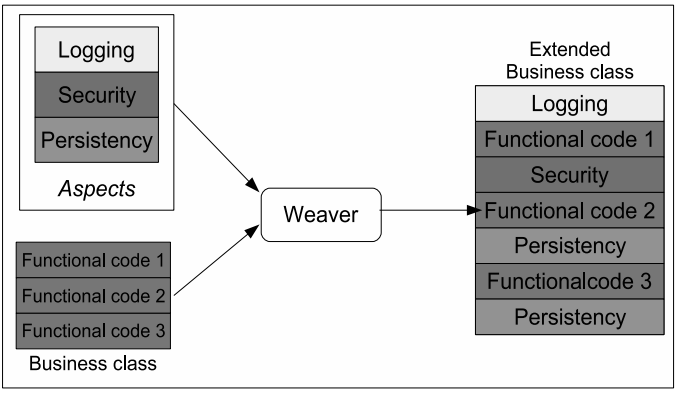


Fig. 3. The weaving process

1. A joinpoint is a simple or structured BPEL activity.
2. The pointcuts are specified on the BPEL document by using XPath.
3. The advice services are BPEL processes implementing the additional behaviour.

We also add to the generated process, a replying activity before each inserted advice service (see figure 4). This activity sends to the client a message called $execute$. This message informs the client about the execution of an additional behaviour. It encapsulates two kinds of information: the identifier of the advice service and its corresponding interaction protocol. This message is necessary since this additional behaviour can require new information exchange involving messages not expected by the client and leading to execution failures. At the level of client implementation, the developer has to handle this type of message: it must extract the interaction protocol of the *advice service* and integrate it in its behaviour. This part is detailed later.

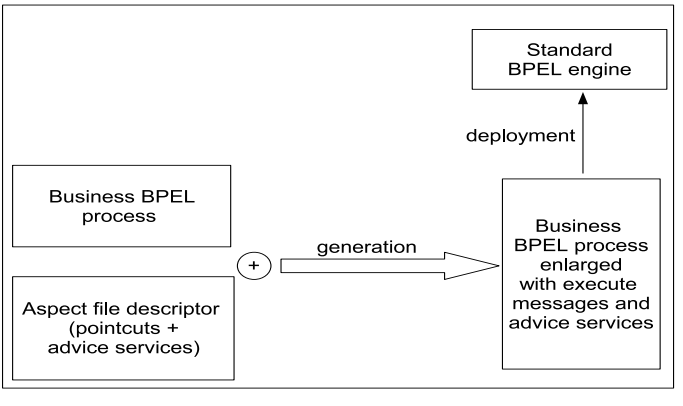


Fig. 4. The extended executable BPEL process

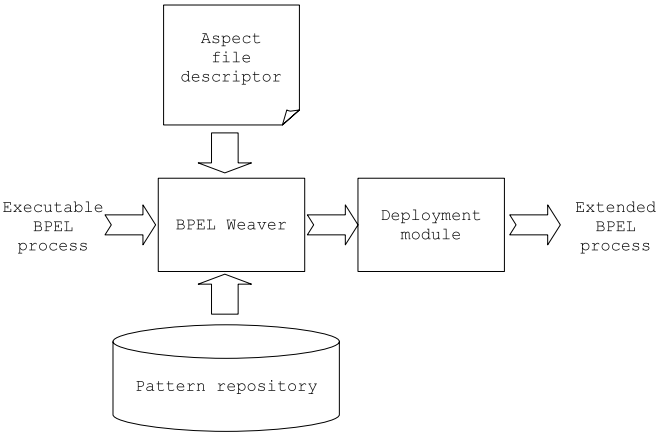


Fig. 5. The extended BPEL generator

3.3 Extended BPEL Generator

These previous concepts are concretized through the architecture of our tool named *extended BPEL generator*. The tool contains the following components (see figure 5):

1. The *BPEL weaver*
2. The *aspect services file descriptor*
3. The *service advice repository* (or the *pattern repository*) which contains the services advices present in the system.
4. The *deployment module* which deploys the extended BPEL process on a standard BPEL engine.

The BPEL weaver takes as input the base BPEL process and the *aspect services file descriptor*. Then, it performs transformations on the base BPEL process syntactic tree. It inserts the actions of sending $execute$ messages and the advices services at the selected joinpoints depending on the kind of the *advice service*. The figure 6 shows the transformations made on the base process

$$receive(ResReq); switch(\{reply(ResResp)reply(error)\})$$

which receives a $ResReq$ message then replies by a $ResResp$ or $error$ message depending on a condition (the $switch$ process). In the case of an around *service advice* (figure 6.d), the specified $joinpoint$ is replaced by the advice service and the $execute$ message replying activity, because we consider that the *advice service* can encapsulate the joinpoint. In the figure, a triangle represents an advice service and Q its corresponding interaction protocol.

3.4 The Extended Interaction Protocol

The extended executable BPEL process interaction protocol is described by an extended abstract BPEL process which integrates the sending of $execute$ messages. The extended

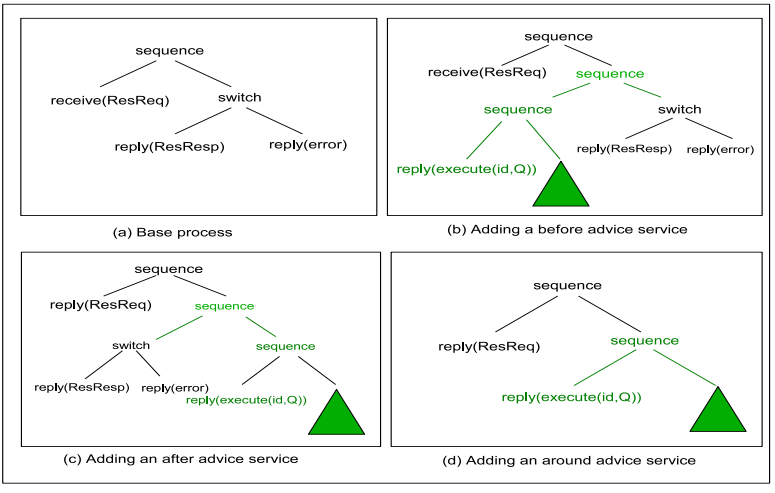


Fig. 6. Syntactic transformations on the base executable BPEL process

interaction protocol is generated from the base BPEL process and the *aspect service file descriptor* based on the defined pointcuts and the type of advices (before, after or around).

The generation process performs transformations on the base abstract BPEL process syntactic tree. It inserts the action of sending $execute$ messages in the selected joinpoints depending on the kind of the *advice service* (figure 7). The *execute* messages contain only the identifier of the *advice service* id. The interaction protocol corresponding to that id is sent to the client at runtime. In this way, the advice service can be

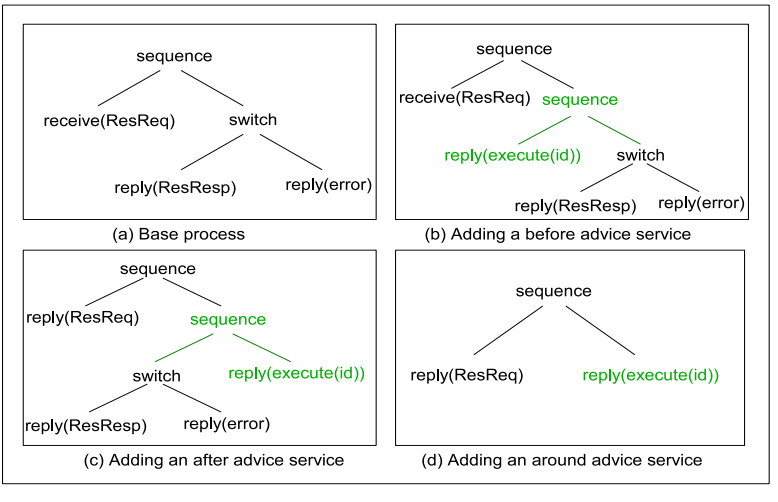


Fig. 7. Transformations on the syntactic BPEL process tree

changed at any time without requiring a new publication. Since we have chosen to let unchanged the BPEL engine, the weaver acts at deployment time.

3.5 The New Operational Rules

In order to take into account the special nature of the message $execute$ we modify the operational rules related to messages.

$$\forall m \in M \ \ *o[m] \xrightarrow{\chi} *o[m] \quad \text{with} * \in \{?,!\}$$

$$\forall m \in M \setminus \{execute\} \ \ *o[m] \xrightarrow{*m} \texttt{empty} \quad \text{with} * \in \{?,!\}$$

$$!o[m] \xrightarrow{!execute(id)} WaitAdvice(id) \tag{1}$$

$$WaitAdvice(id) \xrightarrow{id.\surd} empty \tag{2}$$

The two first rules are similar to those presented in subsection 2.1. In the case of sending an $execute$ message, the automaton evolves to an intermediary state named $WaitAdvice(id)$ (rule 1). $WaitAdvice(id)$ waits for the termination of the *advice service* identified by id. When *advice service* id terminates, $WaitAdvice(id)$ state executes $id.\surd$ and becomes $empty$ process (rule 2). In words, these two rules mimic the synchronisation corresponding to a procedure call.

3.6 The Dynamic Client Interpreter

In order to communicate with change-prone BPEL processes, we extend the previous client interpreter. The new client has to achieve the following tasks:

1. When the client receives an $execute(id)$ message, it has to extract the *advice service* interaction protocol (identified by id) and generates its corresponding server and client automaton.
2. It simultaneously executes the client automaton of the main process and its *advice clients* automata.
3. It makes synchronisation between the main client TA and the advices clients TA on the termination of service advices execution.

Furthermore, the generation module of the dynamic client interpreter also integrates new operational rules for sending and receiving in order to handle the $execute(id)$ messages.

3.7 Execution Scenario

Let us consider the abstract BPEL process defined before. If we want to dynamically add an authentication process before the $switch$ process, the extended abstract BPEL process has to integrate a sending $execute(id)$ message process before the $switch$ process.

$$?o[ResReq]; !execute(id); switch(!o[ResResp], !o[error])$$

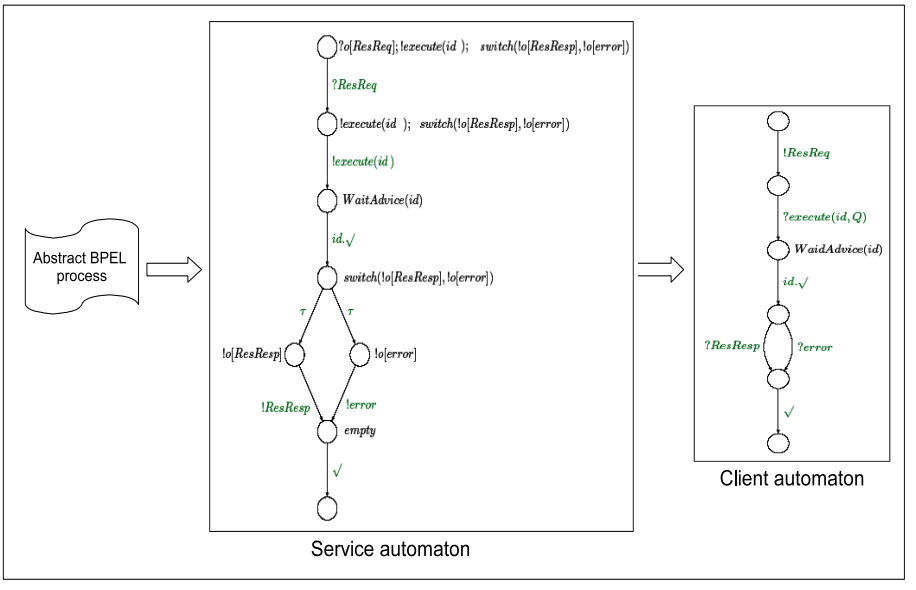


Fig. 8. Adaptable service and client automata

At execution time, the dynamic client interpreter downloads the extended abstract BPEL specification. Then, it generates the corresponding service automaton based on the operational rules previously defined. Afterwards, based on the service automaton and the interaction relation, our client generates the client automaton and begins its interpretation. Figure 3.7 shows the generation process.

When the client receives an $execute(id, Q)$ message, it extracts the abstract BPEL *advice service* process from the message. In our example, the *advice service* is an authentication process which abstract BPEL specification is:

$$!o[authDataRequest]\ ;\ ?o[authDataResp]$$

This process sends an authentication data request to the client asking for authentication data, receives these data then performs some actions to authenticate the user not represented here for simplicity. The client generates the corresponding advice client automaton, associates with the received id and begins its execution (see figure 9 (left part) where states in grey represents the current execution step).

When the *advice client id* terminates, the client synchronises the two automata: it deletes the *advice client*, performs the $id.\surd$ action and continues the execution of the main client automaton (see figure 9 (right part)).

4 Related Work

Formal specification and verification of Web services. Some proposals have recently emerged to formally describe WSs, most of them are grounded on transition system

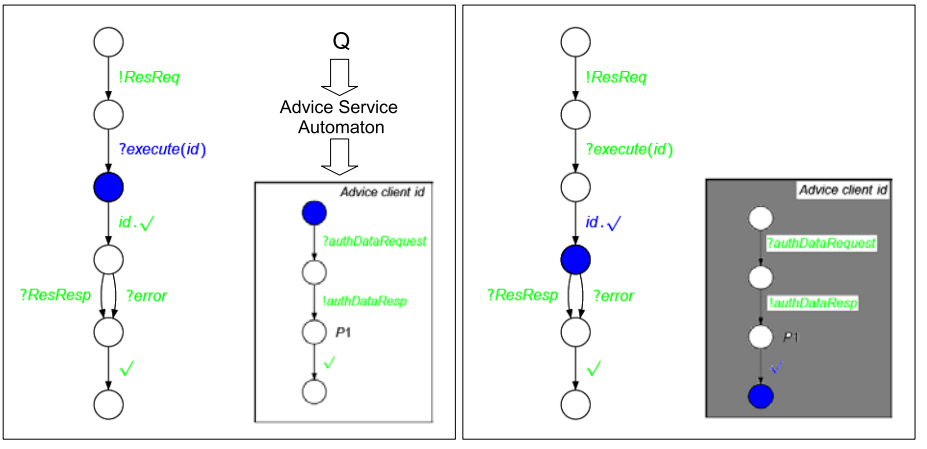


Fig. 9. Reception of an $execute(id, Q)$ message (left) and the termination of an advice service (right)

models (Labelled Transition Systems, Petri nets, etc.) [12,6]. The platform WSAT [8,9] enables designers of a Web service composition to check properties expressed by LTL formulas with SPIN tool. The formal semantics is obtained by gluing patterns for each BPEL construction. One pattern is connected from its final state to the initial state of next pattern according to the BPEL description with local transitions. This work does not cover the time features and it focuses only on message exchanges: the conversation is obtained by a *virtual watcher* that is supposed to record all messages sequences sent by each peer enrolled in the composition.

Another research about Web services formal semantics is based on a BPEL to Finite State Processes (FSP) translation [7]. This work lies on message sequence charts and the core of the verification mechanism consists to check trace equivalence. Again, the time features of the specification are not taken into account.

The work [24] uses the notation CRESS (Chisel Representation Employing Systematic Specification) to formalise Web services. This model presents two main advantages: automatic translation into formal languages for analysis as well as into implementation languages for deployment. Then the CRESS specification is translated into LOTOS and analysed with tools like TOPO, LOLA and CADP. Again, the temporal aspects are not present.

These different contributions share with our approach the design of a formal semantics for Web services. However they study the BPEL execution process and not the interaction protocol, they do not include the time features of BPEL and they perform component verification whereas we perform component synthesis.

AOP and Web services. In [4] and [5], the authors define specific AOP languages to add dynamically new behaviours to BPEL processes. But, neither of these approaches address the client interaction issue. The client has no mean to handle the interactions that can be added or modified during the process execution.

The Web Service Management Layer (WSML) [25] is an AOP-based platform for WSs that allows a more loosely coupling between the client and the server sides. WSML handles the dynamic integration of new WSs in client applications to solve client execution problems. WSML dynamically discovers WSs based on matching criteria such as: method signature, interaction protocol or quality of service (QOS) matching. In a complementary way, our work proposes to adapt a client to a modified WS.

5 Conclusion

In this paper, we proposed a solution based on AOP and process algebra to handle dynamic changes in the context of Web services. We extended our previous AOP approach to support BPEL processes and to handle interaction issues. We also use process algebra formalism to specify change-prone BPEL processes and generate dynamic clients.

As future works, we want to take into account the client execution context. We also want to formally handle the aspect interactions issue (aspects applied at the same joinpoint). Finally, we plane to improve the current ASW prototype as proof-of-concepts.

References

1. AspectWerkz. AspectWerkz 2, http://aspectwerkz.codehaus.org
2. Ben Hmida, M., Tomaz, R.F., Monfort, V.: Applying AOP Concepts to Increase Web Services Flexibility. In: Proceedings of the International Conference on Next Generation Web Services Practices (NWESP 2005). IEEE Computer Society, Los Alamitos (2005)
3. Bergstra, J.A., Klop, J.W.: Process Algebra for Synchronous Communication. Information and Control 60(1-3), 109–137 (1984)
4. Charfi, A., Mezini, M.: Aspect-Oriented Web Service Composition with AO4BPEL. In: Zhang, L.-J., Jeckle, M. (eds.) ECOWS 2004. LNCS, vol. 3250, pp. 168–182. Springer, Heidelberg (2004)
5. Courbis, C., Finkelstein, A.: Weaving Aspects into Web Service Orchestrations. In: IEEE International Conference on Web Services (ICWS 2005), pp. 219–226. IEEE Computer Society Press, Los Alamitos (2005)
6. Ferrara, A.: Web Services: a Process Algebra Approach. In: 2nd international Conference on Service Oriented Computing, ICSOC 2004, pp. 242–251. ACM Press, New York (2004)
7. Foster, H., Uchitel, S., Magee, J., Kramer, J.: Model-based Verification of Web Service Compositions. In: 18th IEEE International Conference on Automated Software Engineering (ASE 2003), pp. 152–163. IEEE Computer Society, Los Alamitos (2003)
8. Fu, X., Bultan, T., Su, J.: Analysis of Interacting BPEL Web Services. In: 13th International Conference on World Wide Web, WWW 2004, pp. 621–630. ACM, New York (2004)
9. Fu, X., Bultan, T., Su, J.: WSAT: A Tool for Formal Analysis of Web Services. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 510–514. Springer, Heidelberg (2004)
10. Haddad, S., Melliti, T., Moreaux, P., Rampacek, S.: Modelling Web Services Interoperability. In: Sixth International Conference on Enterprise Information Systems, ICEIS 2004, pp. 287–295 (2004)
11. Haddad, S., Moreaux, P., Rampacek, S.: Client Synthesis for Web Services by Way of a Timed Semantics. In: Eighth International Conference on Enterprise Information Systems, ICEIS 2006, pp. 19–26 (2006)

12. Hamadi, R., Benatallah, B.: A Petri Net-based Model for Web Service Composition. In: Proceedings of the 14th Australasian Database Conference, ADC 2003. CRPIT, vol. 17, pp. 191–200. Australian Computer Society (2003)
13. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
14. IBM, BEA Systems, Microsoft, SAP AG, and Siebel Systems. Business Process Execution Language for Web Services version 1.1., http://www.ibm.com/developerworks/library/specification/ws-bpel/
15. JBoss. JBossAOP, http://www.jboss.org
16. Juric, M.: BPEL and Java, http://www.theserverside.com/articles/article.tss?l=BPELJava
17. Juric, M.: Business Process Execution Language for Web Services BPEL and BPEL4WS, 2nd edn. Packt Publishing (2006)
18. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.-M., Irwin, J.: Aspect-Oriented Programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
19. Laddad, R.: AspectJ in Action: Practical Aspect-Oriented Programming. Manning Publications Co. (2003)
20. Milner, R.: Communication and concurrency. Prentice-Hall, Inc., Englewood Cliffs (1989)
21. Spring. Spring AOP Platform, http://www.springsource.org/
22. Staab, S., van der Aalst, W., Benjamins, V., Sheth, A.P., Miller, J.A., Bussler, C., Maedche, A., Fensel, D., Gannon, D.: Web Services: Been There, Done That? IEEE Intelligent Systems 18(1), 72–85 (2003)
23. Tomaz, R.F., Ben Hmida, M., Monfort, V.: Concrete Solutions for Web Services Adaptability Using Policies and Aspects. Int. J. Cooperative Inf. Syst. (IJCIS) 15(3), 415–438 (2006)
24. Turner, K.J.: Formalising Web Services. In: Wang, F. (ed.) FORTE 2005. LNCS, vol. 3731, pp. 473–488. Springer, Heidelberg (2005)
25. Verheecke, B., Cibran, M.A., Jonckers, V.: AOP for Dynamic Configuration and Management of Web Services. In: Jeckle, M., Zhang, L.-J. (eds.) ICWS-Europe 2003. LNCS, vol. 2853, pp. 137–151. Springer, Heidelberg (2003)
26. W3C. Web Services Description Language (WSDL) 1.1. W3C Note (March 15, 2001), http://www.w3.org/TR/wsdl
27. W3C. XML Path Language (XPath) Version 1.0. W3C Recommendation (November 16, 1999), http://www.w3.org/TR/xpath

Formal Reasoning about Software Object Translations*

Vladis Berzins[1], Luqi[1], and Peter M. Musial[1,2]

[1] Computer Science Department, Naval Postgraduate School,
1411 Cunningham Rd., Monterey, CA, 93943, USA
[2] Department of Computer Science, University of Puerto Rico, Rio Piedras,
P.O. Box 23328, San Juan, PR 00931, USA

Abstract. In this work we examine the problem of verifying translations from outputs of one system to the inputs of another system, which we refer to as the *output-to-input translation problem*. We present a formalization of this problem along with a verification mechanism based on constraint logic programming. Composition of systems is an important issue in the software reuse domain, and has applicability in other areas of software engineering such as transformation of information from one phase of the development process to another. Some challenges are to verify the translation mechanisms that may be needed to connect independently designed components and assess to what degree is the consumer component functionality enabled after the composition takes place. To this end we use constraint logic programming modeling techniques. Our model allows formalization of the translation problem in the form of constraints and relations between the outputs and the inputs of involved components. Since CLP tools are computationally expensive, we identify characteristics of translation problems for which our technique is practical. We conclude with an application of our translation framework within the Documentation Driven Software Development methodology.

1 Introduction

In this work we address a problem originating from the domain of software component reuse in the design of complex systems, where a system may be composed of complex subsystems. To reduce implementation costs and improve reliability, design of such systems may incorporate existing software libraries or complete subsystems. In practice software reuse is not limited to just matching component interfaces, and will potentially require translation of outputs of the producer component to the inputs of the consumer component. The challenge is that the translation can potentially involve merging outputs via some sequence of operations. In this case it is important to verify that the translation enables functionality of the consumer software component while supporting the outputs of the source software component.

Therefore, the context for this work is conceptual verification of certifiable systems where there exists a prior version of the system that has been certified correct and one part of it is being updated and the system constraints are known. For example, it may be required to replace some component due to new performance demands or since it

* This work is sponsored through AOR grant for development of the DDD framework. In addition P.M. Musial is funded through the NRC Postdoctoral Fellowship.

C. Choppy and O. Sokolsky (Eds.): Monterey Workshop 2008, LNCS 6028, pp. 43–58, 2010.

became obsolete and unsupported. The update must be verified correct in order for the next version to obtain certification as well. The verification of the updated system does not eliminate the need for testing as the replaced component may contain implementation errors, such as memory overflows, timing faults, etc. The goal of this work is to provide a sound verification framework that validates design of the upgrade, under the assumption that the new component meets its specifications, as expressed by the constraints associated with the component. The verification of the translation targets architecture level faults and should help reduce system integration problems.

1.1 Validating Component Compositions

The problem of validating the composition of components is an active research area. A previously studied approach to the problem is to use an object-oriented modeling language and graphical tools to define and reason about component composition. For example, GenVoca generators [1] are used to synthesize software systems by composing components from reuse libraries. In GenVoca, components are parametrized program transformations that encapsulate consistent data and operation refinements. In [2] it is shown that GenVoca can be used to validate composition of components. However, the assumption in [2] is that the compatible components implement same abstract interface. In our work we do not make this assumption. Moreover, the degree to which the consumer component is enabled as a result of the composition is not being measured in [2], which we measure in this work by computing the range of the output of the consumer component as compared to the output produced before composition takes place. Meaning, constraint logic program can provide information about the possible solution space for the constraint program that defines the new composition based on the set of given inputs, which can be compared to the range of outputs produced for the same set of inputs by the consumer component either on its own or in an existing system configuration.

Another way to approach the problem is through reasoning about interface matching and validation based on the detailed knowledge of the component's code, as was done in [3]. Specifically, [3] presents an approach to modeling components and component composition, which incorporates the notion of communication between individual components in the composition. Verification of composition is performed on interactions between component interfaces. Again, translation is not considered. An interesting aspect of [3] is that the proposed model can check if a specific (transition) path in the composition is reachable. However, the extent of reachable paths is not computed.

In [4] a conceptual framework is presented for software component definition, validation, and composition. This framework is dubbed ComDeValCo and it approaches the problem of component composition by structuring system components as a library of components. The verification and validation methods for composition are not well detailed.

The problem of translation in the context of web applications is examined in [5], where a formal model is presented for providing a mapping between two independent web modules. The key contribution of [5] is that they remove the lexicographical mapping requirements between components and allow the designer to choose which outputs and inputs should be connected. In addition, [5] presents a mechanism for code generation from the mapping. The translations are mathematically specified through abstract

state machines. However, the mapping is a direct mapping (one-to-one) and authors of [5] do not provide a verification mechanism for the mapping. In our work we relax the one-to-one mapping requirement and allow components in the composition to be connected via an intermediate translator.

The above works approach the problem of component composition from a systems level. There are higher level approaches to component composition that involve reasoning at a more abstract level of automata such as Timed Input/Output Automata [6]. However, it is often the case that an abstract automata representation of components may be difficult and time consuming to extract from existing implementations. In our approach we attempt to provide a bridge between the abstract and low level system specifications, where our framework will attempt to verify component translation based on the amount of information given. Clearly, the more detailed information is provided about the components, domain of inputs, range of outputs, and the translation functionality, the more informative the answers will be.

Type-checking is another classic approach that can be used to verify matching between outputs and inputs in the composition. However, static type-checking is not sufficient to verify values of the data based on examination of the code, for example checking bounds on array indexes. Dynamic type-checking can be used to verify variable assignments, but such an approach is usually used at runtime and to the best of our knowledge cannot always be used as a proactive mechanism [7].

In this work we abstract the problem of component composition to its functional behavior. Specifically, we relax assumptions that component code is available, rather we assume that only constraints on the behavior (functionality) of the components is known along with domain information of the input and outputs. These constraints characterize the slots in the architecture that are to be filled by the components in question. The constraints represent the standards imposed by the architecture on "plug-compatible" components. These standards typically do not completely characterize all the details of the behavior of acceptable component, although ideally they should be strong enough to guarantee that any component that meets the standards will enable the architecture to perform its intended functions.

Clearly the more information about components is available the more precise guarantees can be provided about the composition. We do not assume that there will be a strict interface matching between components and that the use of a non-trivial translation layer may be necessary. We are interested in verification of the composition by measuring the degree of enabled functionality of the consumer component after the composition takes place.

1.2 Constraint Logic Programming

Constraint logic programming (CLP) is a programming paradigm that allows expressing logical relations among several unknown variables, where each variable accepts values from some domain. For example (from [8]), assume placement of a square and a circle in a two dimensional plane, where the relation between these two objects is that the circle should be contained within the square. Note that the size of these objects and the location of the circle within the square are not specified. One may add additional constraints to this system that describe specific ratios, distances from the borders, or

add additional objects and introduce constraints on the relations between all objects. A CLP solver is a tool that provides an answer whether the constraints expressed by a CLP program are satisfiable, unsatisfiable, or undecidable.

CLP has been used to successfully model complex problems from various domains, such as design of analog and digital circuits, civil and mechanical engineering, finance, assembly line optimization, building visual language parsers, and many others (for a comprehensive survey of CLP models we refer the reader to [9]).

Solving a CLP program involves the problem of constraint satisfaction [10], which can be a computationally difficult problem. This means that for certain classes of problems constraint satisfaction requires exponential time with respect to the number of variables, for example all problems that are reducible to the 3SAT problem. The good news is that many practical problems can be defined in terms of constraints over finite domains (including some problems from the boolean domain). Programs that require infinite domains are efficiently solvable if constraints can be specified as linear constraints on integer variables; the same is not true for nonlinear constraints. Continuous domains are common in real-world problems, where for this class of problems there are efficient solutions for constraint satisfaction when constraints are expressed as linear inequalities forming a convex region – linear programming. For a brilliant presentation of the constraint satisfaction problem we direct the interested reader to [10]. For practical pointers on modeling decisions that make a solution to a constraint logic program terminate quickly we direct the reader to [11]. Advances in the research on CLP solvers and increasing computational power of computers makes CLP an attractive method for solving problems in a wide range of domains, including software engineering. However, in the context of this work we will introduce restrictions to make our approach practical.

In this work we are interested in validating the translation between two software components, but also in assessing size of the solution space. Doing so can give insight about the functionality being enabled of the target software component. When the size of the solution space produced by the consumer component after the composition is same as the size of the solution space prior to the composition, then the composition preserved the functionality of the consumer component, else some functionality has been restricted.

Document Structure. In Section 2 we present a general framework for component translation. At this point we abstract from software components and present our framework in terms of generic components that have inputs, outputs, and functionality. In Section 3 we introduce a general model within which the translator is defined in terms of constraints and relations between outputs and inputs, and we present some necessary conditions for validation of a given translation. Various constraint domains are discussed in Section 4. In Section 5 we examine application of our framework within Documentation Driven Software Development framework, by augmenting an Open Architecture [12]. We conclude with final remarks and point out future research directions in Section 6.

2 Formal Modeling of Component Translations

Figure 1 depicts a composition of two components that is accomplished by use of a translation of outputs of the source component to the inputs of the target component.

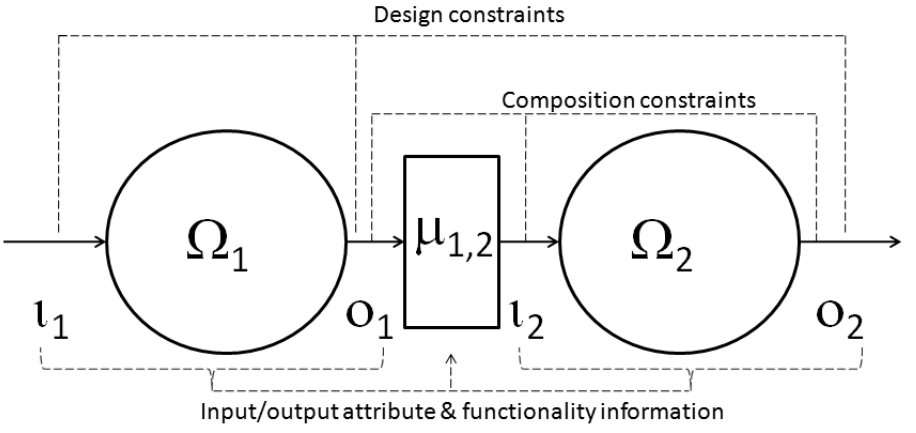


Fig. 1. A general representation of the translation process from outputs of one component to inputs of another

Specifically, outputs of Ω_1 are mapped via a logical relation to the inputs of Ω_2 through the mapping $\mu_{1,2}$. (In the configuration that is being replaced, Ω_2 is connected directly to some other producer component.)

The objective is to verify to what degree the translation $\mu_{1,2}$ enables the functionality of the consumer component. Specifically, the range of output of the consumer component prior to new composition configuration is produced by the functionality of consumer component based on the allowed domain of system inputs. The new composition configuration and the use of a translator may restrict the domain of inputs of the consumer component and consequently reduce the range of its outputs. It may be possible for such differences to be measurable and used as indicators of the degree of enabled functionality.

2.1 Assumptions

The primary assumption is that the translation is unidirectional, where the data flows from the producer component to the consumer component. This is done for presentation purposes and the bidirectional extension is straightforward.

We assume that the components used in the translation are well-formed, with inputs and outputs that are bound to some specific set of types. The well-formedness assumption restricts components to ones that have a fixed functionality and fixed protocol for interacting with their connections. Note that our assumptions are not very restrictive and do not rule out all models that may exceed descriptive powers of the specific CLP language used or may exceed capabilities of the accompanying solver. We provide a short survey of CLP languages and their capabilities and implementations in Section 4.

2.2 Notation and Model

In this section we describe our model along with definitions of mathematical notation and symbols used to describe components and their functionality, as well as the meaning of

the translation. In order to avoid notational clutter, whenever we refer to a component, or component's inputs or outputs outside of the composition we forgo the use of subscripts.

A component Ω is represented as a tuple from $\iota \times o \times R \times C$. The elements of ι and o are tuples with at least one field, more formally $o = \langle o_1, o_2, \ldots, o_n \rangle$ and $\iota = \langle \iota_1, \iota_2, \ldots, \iota_m \rangle$, where n and m represent the number of outputs and inputs respectively. Each field of ι, respectively o, is defined over a domain of some fixed type. R is a mapping from inputs to outputs, such as $R : \iota \overset{\{r_1, r_2, \ldots, r_n\}}{\rightarrow} o$, where $\{r_1, r_2, \ldots, r_n\}$ represents the set of relations that map component inputs to each of the outputs. Components with internal states are modeled by adding an extra input and output for each state variable, so that state variables can be used in constraints. R is modeled as a relation to accommodate possibly nondeterministic component behavior. C is a set of constraints defined over inputs and outputs. Constraints are terms that define desired properties of the inputs and outputs. There should be at least one constraint on each field of ι and o. For instance, assume that the first field of input ι accepts any number from the domain of natural numbers, in this case the minimal constraint on the first field of the input is: $\text{domain}(\iota(1)) = \mathbb{N}$. In addition, C contains constraints that define relations between individual inputs and outputs that realize the component transformation. For example, a component may have two outputs where one is a checksum of the other, hence a constraint would have to be added that describes this relation. Note that $\text{range}(o(\cdot))$ can be defined directly as a set as it was done for $\text{domain}(\iota(1))$, or it can be defined as a set computed over the relation r_i for some i between 1 and n. Specifically: $\text{range}(o(i)) = \{y : \exists x_1, \ldots, x_m (\forall (1 \leq j \leq m \Rightarrow x_j \in \text{domain}(\iota(j))) \wedge y \in r_i(\langle x_1, \ldots, x_m \rangle))\}$, where $1 \leq i \leq n$.

A translation $\mu_{1,2}$ is a mapping from the outputs of Ω_1 to the inputs of Ω_2, and is represented by a tuple of the form $\iota \times o \times R$, where fields of this tuple are defined similarly as above. In addition, $\text{range}(o_\mu(i))$ is defined similarly to $\text{range}(o(i))$ (see the preceding paragraph).

Therefore, given $\mu_{1,2}$, the definitions of Ω_1 and Ω_2, and the set of system constraints, verification of the translation requires that we check at least the following goal conditions.

$$\text{range}(o_1(i)) \subseteq \text{domain}(\iota_{\mu_{1,2}}(i)), 1 \leq i \leq n_{\mu_{1,2}} \tag{1}$$

$$\text{range}(o_{\mu_{1,2}}(j)) \subseteq \text{domain}(\iota_2(j)), 1 \leq j \leq m_2 \tag{2}$$

The above conditions reflect the requirement that the domain of the mapped outputs of Ω_1 via some combination of operators to the inputs of Ω_2 falls into the acceptable domain. Hence, ensuring that the output produced by the mapping of outputs of Ω_1 will be supported by inputs of Ω_2.

The goals can also include assertions that describe the intended cooperation between the components. For a simple example consider a secure internet connection that encrypts data while it is in transit. In such a case component Ω_1 should act as an encoder, component Ω_2 a decoder, and an appropriate additional goal would be that the two components compute transformations that are inverse of each other, which can be expressed as the assertion $o_1 = \iota_2$.

3 Translation Verification

Thus far we presented a general framework that defines the translation from outputs of the producer component to the inputs of the consumer component. The remaining activity is the verification of the translation. This section presents a method for verification of μ by using a *constraint logic programming* [9] framework. The discussion that follows presents an approach for a general, theoretical CLP solver, and not for any specific CLP implementation. However, this general approach is used as a guideline for implementation of CLP solvers.

A constraint logic program can be defined in terms of tuple $\langle G, S\rangle$, where G is a set of constraint goals, and S is a set of constraints. In our model, a minimal set of constraint goals is represented by conditions (1) and (2) in the preceding section. These goals ensure that the communication between the producer component and the translation, and the translation and between the consumer component allow unrestricted information flow. Therefore, our aim is to prove each of the terms in G to be true. The set S represents the set of conditions under which the set of goals has to hold. Therefore, constraints in S are assumed to be true and represent system requirements that have been proved correct beforehand. The solver verifies constraints from G by moving these that evaluate to true to the set S. A system is satisfiable when all constraints in G evaluate to true and the set G is empty at the end of the prove; otherwise, the system is unsatisfiable. Therefore, the solver checks whether all constraints in $G(X)$ and $S(X)$ hold true for all interpretations of system variables in the set X, where X represents the set of system variables. In a well formed goal, the only free variables correspond to the inputs and outputs of the components and the translation itself. Per our assumption each field of each input and output is associated with at least one constraint. Hence all of the free variables that appear in G also appear in S. When the solver is successful the system $\langle G, S\rangle$ is satisfiable, and since G has no free variables other than those in X, the goal assertions must be true for any scenario in which the constraints hold. We conclude that within the composition the ranges of outputs are subsets of domains of the inputs – as desired. (Recall that a constraint system is undecidable when the termination condition cannot be reached, see [13] for debugging techniques.)

Hence, using our model, we convert conditions (1) and (2) to the following set of goals:

$$G = \bigcup_{1\le i\le n_{\mu_{1,2}}} \{\text{range}(o_1(i)) \subseteq \text{domain}(\iota_{\mu_{1,2}}(i))\} \cup \bigcup_{1\le j\le m_2} \{\text{range}(\mu_{1,2}(j)) \subseteq \text{domain}(\iota_2(j))\} \tag{3}$$

The set S is simply:

$$S = C_1 \cup C_2 \cup C_{composition\ constraints} \cup C_{design\ constraints} \tag{4}$$

Where C_1 and C_2 denote the constraints associated with Ω_1 and Ω_2, respectively. $C_{composition\ constraints}$ and $C_{design\ constraints}$ (see Figure 1) represents any additional constraints on the composition of the two objects. For example, the system design may impose that the outputs of Ω_2 have a specific relation to outputs of Ω_1 or inputs

of Ω_2. Therefore, these constraints enforce the semantics of the transformation implemented by the composition of the two components. In practice, the developer does not need to be specific about which constraints belong to G and which belong to S. Rather, the developer in addition to the constraints depicted in (3) identifies all system constraints and the logical relation between the system variables.

The following a very simple example illustrates a situation in which a constraint model allows us to verify semantics of the translation that goes beyond simple type checking. A producer component produces an integer value as an output, and the consumer component accepts an integer value and an input – these requirements represent constraints C_1 and C_2. However, the values output by producer are opposite of what is expected by the consumer component – an example of a composition constraint, $C_{composition\ constraints}$ (i.e., translation implements the following relation $o_1(i) = -1 \times \iota_2(j)$ for some appropriate indexes i and j). The goal constraints are trivial in this example and basically require that outputs and inputs are constrained to the integer domain. Such condition is easy to verify using a constraint model, but may be difficult to verify using other methods.

Assessing The Solution Space. The CLP program as defined by (3) and (4) will produce a set of terms. If the solution set to our program is an empty set, meaning the solution space is empty, then the given translation is too limiting. Otherwise, the degree to which the functionality of Ω_2 is enabled can be assessed by comparing the size of the produced solution space to the size of the domain of ι_2. Note that the range of o_2 represents the solution space produced by Ω_2 in isolation from the new composition configuration, where all values in the range of o_2 were produced as a result of some function of Ω_2. CLP solvers support verbosity levels that allow printout of the reachable solutions and the decisions made along the way. The solution trace will depend on the characteristics of the program and the CLP language/solver that is being used. This means that the composition can be re-evaluated against the system design requirements and validated against the trace printout of the reasoning leading to possible solutions. Trace analysis could be performed by a system designer or possibly be automated. Such analysis can uncover whether the conceptual output of the system is correct or not. If the output is not correct, then the analysis can help to identify which constraints or functionality of Ω_1 or $\mu_{1,2}$ are responsible for unnecessarily restricting the desired behaviors of the newly composed system. (For additional information on debugging constraint programs please see [13].)

4 Extent of Our Approach

CLP Scheme [14,9] defines classes of languages denoted as CLP($\mathcal{X}$), where $\mathcal{X}$ is a pre-interpretation defining the primitive constraints, functions, and their interpretation. Specifically, this description contains the following information [15]: *signature*, which defines a set of function and predicate symbols and associates arity with each symbol, hence defining the terms and primitive constraints of the constraint language, *domain*, which defines the intended interpretation of the constraints, *theory*, which describes the logical semantics of the constraints, and *solver*, which is a description of a mechanism that can determine where a program described in the language is satisfiable,

unsatisfiable, or undecidable. Next we list the more prominent classes of domain constraints. The following summary is based on the CLP survey [9].

- $\mathcal{R}$: The signature consists of linear arithmetic operators, and constants 0 and 1. The domain is the real number set. Hence CLP($\mathcal{R}$) is a language that supports arithmetic operations over real numbers.
- $\mathcal{R}_{Lin}$: a constraint domain defined similarly to $\mathcal{R}$ with arithmetic operation $*$ removed. Basically, limited to linear inequalities.
- $\mathcal{Q}_{Lin}$: defined similarly to $\mathcal{R}_{Lin}$, but restricted to the domain of rational numbers only.
- $\mathcal{FT}$: the signature contains a collection of constant and function symbols and the predicate $=$. The domain is a set of finite trees. The primitive constraints are equations between terms. Basically, it is the Herbrand constraint domain, i.e. based on equations on the algebra of finite terms.
- $\mathcal{RT}$: defined similarly to $\mathcal{FT}$, where the difference is that the domain includes infinite trees.
- $\mathcal{FEAT}$: the signature consists of a binary predicate symbol $=$, a set of unary predicate symbols (called sorts), and a set of binary predicate symbols (called features). The domain is a set of trees (not necessarily finite), where nodes are sorts and edges are features. Hence, the constraint domain is defined over feature trees.
- $\mathcal{WE}$: is a language is defined over strings and characters with the concatenation and equality operations. Hence, it is a constraint domain of equations of strings.
- $\mathcal{BOOL}$: the signature consists of 0, 1, and operators $\neg$, $\wedge$, $\vee$, $\otimes$, $\Rightarrow$, and $=$. The domain is limited to two values: true and false. Hence, this is a two-valued Boolean constraint domain.
- $\mathcal{FD}$: is a constraint domain that is referred to as the finite domain. The signature contains operators $+$, $=$, $\neq$, and $\leq$, and the domain is restricted to a bounded set of integer values.

The literature contains other constraint domains (see [9]). Tools exist that support the above languages (although not all features may be supported). Some examples of CLP tools include BNR-Prolog [16], CAL [17], CHIP [18], CLP($\mathcal{R}$) [19], Prolog family [20], RISC-CLP(Real) [21], and Choco [8]. Each of these tools supports one or more $\mathcal{X}$ and provides solvers that are able to answer if a CLP program is satisfiable, unsatisfiable, or is undecidable under supported $\mathcal{X}$. The Choco solver is especially interesting since it is a constraint programming system that can be used to define a software architecture for variable domains, constraints, propagation and tree search and implements the basics of a constraint system.

In addition to the above CLP languages, there exist other more exotic CLP related languages such as: REF-ARF [22] which is essentially a procedural language and it supports non-determinism because of constraints used in conditional statements. REF-ARF also supports statements such as $x = x + 1$, where such statements are treated as constraints of the form $x_{i+1} = x_i + 1$. In [23], it is shown how to translate concurrent systems with infinite state spaces to CLP programs while preserving the semantics in terms of transition sequences. Another CLP relative is the Oz [24] language, which contains most of the concepts of the major programming paradigms, including logic, functional (both lazy and eager), imperative, object-oriented, constraint, distributed,

and concurrent programming. Oz has both a simple formal semantics and an efficient implementation – The Mozart Programming System [25]. Oz is a concurrency-oriented language that makes concurrency both easy to use and efficient.

To sum up, there is a rich set of CLP languages that can be used to describe and reason about component compositions. CLP solvers are becoming more powerful and are able to provide solutions to a plethora of practical problems. Choice of the specific tool is based on the nature of problem that needs verification and the functionality of the convolved components.

5 Application Examples

Software systems in civilian and military domain are increasing in complexity and have ever more significant impact on human safety, financial resources, and national security. Complexity of these systems requires incremental development by design of subsystems which are composed together to yield the complete complex system of systems.

Large scale software systems share any subset of the following properties: long development time, global deployment strategies, mission critical requirements, significant resource demands, timing constraints, high quality and reliability standards, ease of reconfigurability, and interoperability with other systems.

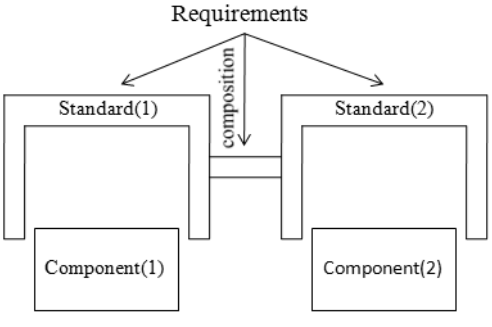


Fig. 2. Open architecture

The key challenges encountered during design of complex systems include: how to generate high quality and high confidence software, how to support system evolution and accommodate changing requirements, how to enable support for variety of stakeholders, and how to improve efficiency and productivity of the development process. The feature that ensures successful development, implementation, deployment, and sustainability is precise documentation.

Documentation Driven Software Development (DDD). The DDD [26] framework is a software engineering methodology that provides assistance for all software life cycle processes, most notably, requirements gathering, quality assurance, design, system evolution and re-engineering, and project management. Each of the software life cycle stages involves communication between stakeholders and the development teams. These two groups share the same objective, but their expertise is in different and sometimes mutually unfamiliar domains. DDD provides mechanisms that allow project information to be effectively communicated between all involved parties, hence providing a bridge between domain of the stakeholders and the domain of developers (which is software design and implementation). Finally, the developers and stakeholders will utilize software and hardware tools during each of the software life cycles. The challenge here is to ensure proper transformation of project requirements, which may be

specified informally, into the formal and mathematical format that is required by the utilized tools. The DDD framework provides mechanisms that help to do just that. Documentation is backbone of the DDD framework. The novel part of the DDD methodology is that all aspects of project information are considered as documentation, which means that documentation is not only limited to natural language text representing system design specifications, manuals, etc. but also includes formal models, knowledge bases, code, simulations, etc. With this definition, the documentation in our approach can provide more effective support for the entire development process.

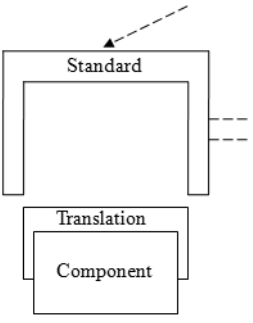


Fig. 3. Component translation wrapper

5.1 Global Architecture

An objective of the DDD framework is to enable systematic construction of reliable software architectures for mission critical systems, particularly with respect to timing constraints extracted from requirements. To this end an open architecture was proposed in [12,27] that allows dynamic system reconfiguration and potentially reduces system testing time. This is achieved by use of standards, requirements/capabilities and environmental assumptions along with components, connections, and constraints (see Figure 2).

As depicted in Figure 2, standards are developed from system requirements. Composition of subsystems is performed on standards. This approach allows system to be tested by examination of standards and their interactions. Implementation of the standard is represented as a plug-in component. Components can be replaced at any time, and the testing needs to be localized to the component/standard interaction. However, authors in [12,27] do not provide detailed explanation of how the interaction between the component and the standard is tested. Our framework can be used to reason about composition of the plug-in and the standard.

5.2 Applications in Open Architecture

The open architecture presented in [12] is an ideal candidate to apply our translation framework. Specifically, the standards used in that context should be well defined entities where functionality and outputs are well documented. A reasonable assumption is that the plug-in component is not an undocumented black-box and partial or complete information about its inputs, outputs, and functionality is known a priori to the developers. However, since standards are inflexible entities, available legacy the plug-in components may be complex to easily modify and its behaviors may not exactly match that of the standard, translation may be necessary.

Figure 3 depicts under the Open Architecture presented in Figure 2 where we use translation in conjunction with standards and components. Under our assumptions, the designer should have ample information to use our framework and test whether the component supports a sufficient range of outputs in order to enable the functionality of the standard, and whether the translation preserves all system constraints. Moreover, our framework enables verification of compatibility of the new component prior to the

installation, hence avoiding system downtime improving confidence in the system after the new component is installed. Furthermore, by forcing the developer to think about the functionality and the composition in terms of the translation and constraints, the testing process for the new component can become more streamlined, where the test cases can be developed based on CLP program – i.e., test outer bounds of the constraints.

5.3 Enumeration Example

This section presents a simple example that is representative of a large class of problems. Specifically, we consider pattern matching via simple enumeration. This type of problem is often difficult to verify for a human, and is commonly found in genetics, security, and logic circuits.

Consider a system consisting of two connected components. Without loss of generality, we can assume that there are two versions of the system, the old version and the new version that is an updated based on the old system – perhaps the old producer component may need to be replaced due to incomparability with the new deployment platform.

Old system. Component one, Ω_1, has one input and two outputs. Component two, Ω_2 has two inputs (outputs are not important). Component one inputs are from the alphabet $\{1, 2, 3, 4\}$, and component one produce 5 output values from the following sets: output one {AB, BA, CA, CB} and output two {A,B,C}. Note that these are the simple constraints on the domains and ranges, i.e., C_1 from equation (4 in Sec. 3). A system constraint is that the only legal inputs to component two are the following pairs: ⟨ AB,C ⟩, ⟨ BA,C ⟩, ⟨ CA,B ⟩, and ⟨ CB,A ⟩ – constraints making up C_2. The meaning of the listed tuples is ⟨ input value to $\Omega_2\iota(1)$, input value to $\Omega_2\iota(2)$ ⟩ . We know that in the old system the component one adheres to the system constraint and implements the following relation: R_{old}: {⟨ 1,⟨ AB,C ⟩ ⟩, ⟨ 2,⟨ BA,C ⟩ ⟩, ⟨ 3,⟨ CA,B ⟩ ⟩, ⟨ 4,⟨ CB,A ⟩ ⟩} – a constraint in $C_{composition\ constraints}$.

New system. We are asked to replace the component Ω_1 with a new component, $\Omega_{1(new)}$, where the replacement has the following specifications: The new component: one input over the alphabet $\{1, 2, 3, 4\}$; and three outputs, two of these are over the alphabet $\{A, B, C\}$, and one over alphabet $\{A, B\}$. Specification for the new replacement states that R_{new}: {⟨ 1,⟨ C,A,B ⟩ ⟩, ⟨ 2,⟨ C,B,A ⟩ ⟩, ⟨ 3,⟨ B,C,A ⟩ ⟩, ⟨ 4,⟨ A,C,B ⟩ ⟩} – this constraint represents $C_{1(new)}$.

The engineer is told that concatenation of some combination of the two outputs should produce the output sequences consistent with the old component. Therefore, the translation element is reduced to a simple string concatenation problem. Where the inputs of to the translation are defined as the outputs of Ω_{new}, and its outputs are defined as inputs of Ω_2. The goal set G in equation (3) defines constraints on domains and ranges of inputs and outputs between Ω_{new} and $\mu_{new,2}$, and $\mu_{new,2}$ and Ω_2. The question is whether this can be validated to be true. Clearly in this case an engineer can preform the manual computation and answer the question accurately and with a manageable degree of effort. However, the same may not remain true as the logic and patterns to match become more complex.

CLP provides an alternative to the manual process. A simple Choco [8] program, see Figure 4, can be written to test various plausible combinations of the outputs and to

verify whether the new replacement component can be used. The answer to our problem is *yes*, where $\Omega_{1(new)}o(2) \cdot \Omega_{1(new)}o(3) \Rightarrow \Omega_2\iota(1)$ and $\Omega_{1(new)}o(1) \Rightarrow \Omega_2\iota(2)$. The designer chose this connection pattern and the solver checked that it does satisfy all of the intended constraints.

```
import static src.choco.Choco.*;
// Import list is truncated in order to improve presentation.
import src.choco.*;

public class TranVer {
   private static IntegerVariable a = makeEnumIntVar("a", 1, 1);
   private static IntegerVariable b = makeEnumIntVar("b", 2, 2);
   private static IntegerVariable c = makeEnumIntVar("c", 3, 3);

   // Defines relation between the inputs to the system and expected input values to Omega 2
   private static Constraint systemTestOmegaTwoInput(IntegerVariable[] tup) {
      return makeExpression(or(
            and(eq(var(tup[0]),var(1)), eq(var(tup[1]),var(a)), eq(var(tup[2]),var(b)), eq(var(tup[3]),var(c))),
            and(eq(var(tup[0]),var(2)), eq(var(tup[1]),var(b)), eq(var(tup[2]),var(a)), eq(var(tup[3]),var(c))),
            and(eq(var(tup[0]),var(3)), eq(var(tup[1]),var(c)), eq(var(tup[2]),var(a)), eq(var(tup[3]),var(b))),
            and(eq(var(tup[0]),var(4)), eq(var(tup[1]),var(c)), eq(var(tup[2]),var(b)), eq(var(tup[3]),var(a)))));   }

   // Defines relation between the inputs to the system and expected input values to Omega 1
   private static Constraint systemTestOmegaOneOutput(IntegerVariable[] tup) {
      return makeExpression(or(
            and(eq(var(tup[0]),var(1)), eq(var(tup[1]),var(c)), eq(var(tup[2]),var(a)), eq(var(tup[3]),var(b))),
            and(eq(var(tup[0]),var(2)), eq(var(tup[1]),var(c)), eq(var(tup[2]),var(b)), eq(var(tup[3]),var(a))),
            and(eq(var(tup[0]),var(3)), eq(var(tup[1]),var(b)), eq(var(tup[2]),var(c)), eq(var(tup[3]),var(a))),
            and(eq(var(tup[0]),var(4)), eq(var(tup[1]),var(a)), eq(var(tup[2]),var(c)), eq(var(tup[3]),var(b)))));   }

   public static void main(String[] args) {

      Solver pb = new CPSolver();
      IntegerVariable omega1in1 =   makeEnumIntVar("Omega1.in(1)", 1, 4); // input to Omega 1
      IntegerVariable omega1out1 = makeEnumIntVar("Omega1.out(1)", a.getBinf(), c.getBinf()); // output 1 of Omega 1
      IntegerVariable omega1out2 = makeEnumIntVar("Omega1.out(2)", a.getBinf(), c.getBinf()); // output 2 of Omega 1
      IntegerVariable omega1out3 = makeEnumIntVar("Omega1.out(3)", a.getBinf(), b.getBinf()); // output 3 of Omega 1
      IntegerVariable omega2in1b1 = makeEnumIntVar("Omega2.in(1).b1", a.getBinf(), c.getBinf()); // input 1 bit 1 of Omega 2
      IntegerVariable omega2in1b2 = makeEnumIntVar("Omega2.in(1).b2", a.getBinf(), b.getBinf()); // input 1 bit 2 of Omega 2
      IntegerVariable omega2in2 =   makeEnumIntVar("Omega2.in(2)", a.getBinf(), c.getBinf()); // input 2 of Omega 2

      // Definitions of problem variables
      IntegerVariable[] varsOmega1 = new IntegerVariable[4];
      IntegerVariable[] varsOmega2 = new IntegerVariable[4];
      varsOmega1[0] = omega1in1; // input to Omega 1
      varsOmega1[1] = omega1out1; // output one of Omega 1
      varsOmega1[2] = omega1out2; // output two of Omega 1
      varsOmega1[3] = omega1out3; // output three of Omega 1
      varsOmega2[0] = omega1in1;
      varsOmega2[1] = omega2in1b1; // input one (bit 1) of Omega 2
      varsOmega2[2] = omega2in1b2; // input one (bit 2) of Omega 2
      varsOmega2[3] = omega2in2; // input two of Omega 2

      // Relation implemented by the translation
      Model m = new CPModel();
      m.addConstraint(eq(omega1out2, omega2in1b1));
      m.addConstraint(eq(omega1out3, omega2in1b2));
      m.addConstraint(eq(omega1out1, omega2in2));

      // Constraints defining relations expected inputs of Omega two
      m.addConstraint(systemTestOmegaTwoInput(varsOmega2));
      // Constraints defining relations expected outputs of Omega one
      m.addConstraint(systemTestOmegaOneOutput(varsOmega1));

      pb.read(m);
      CPSolver.setVerbosity(CPSolver.SOLUTION);

      // Invokes the Choco solver
      if (pb.solve()) {
         // Print values of problem variables when solution is reached
         do {
            for(int i = 0; i < varsOmega1.length; i++) {
               System.out.print(varsOmega1[i].pretty());
               System.out.print("␣");
            }
            for(int i = 1; i < varsOmega2.length; i++) {
               System.out.print(varsOmega2[i].pretty());
               System.out.print("␣");
            }
            System.out.println("␣");
         } while (pb.nextSolution() == Boolean.TRUE);
      }

      if (!pb.isFeasible())
         System.err.println("No␣solutions␣can␣be␣found.␣");
   } }
```

Fig. 4. Choco program verifying translation logic from example given in Section 5.3

5.4 Applications in Plug-in Technology

Proposed framework has applications in the military operations where system safety is often critical. Following is a brief description of one possible scenario where system safety requires verification of component composition.

Component translations are needed in applications where the data are sought for a mission support application that performs *electromagnetic spectrum* (EMS) predictions used for surveillance and analysis of radar and communication signals in areas where military forces will operate. This EMS prediction model has relatively stable data requirements that are provided from various data sources. The EMS application presents the prediction results of the model as graphical charts or as graphical overlays on a map. Figure 5 depicts a majority of the data required by the EMS prediction model and the EMS prediction model outputs.

Our approach allows verification of compatibility of plug-ins with the expected standards of the EMS module. The process verifies the data formats of the various data sources against the data formats supported by the EMS model. If new data feeds become available that are not in a standard format (or are in a new standard format), the developers manually reverse engineer the translating code to derive the data required by the EMS application. Similarly, our approach can be applied while handling the outputs of the EMS module. Verification of plug-in compatibility prior to implementation of the needed translations has potential to reduce risk and production cost, and increases confidence in the final system.

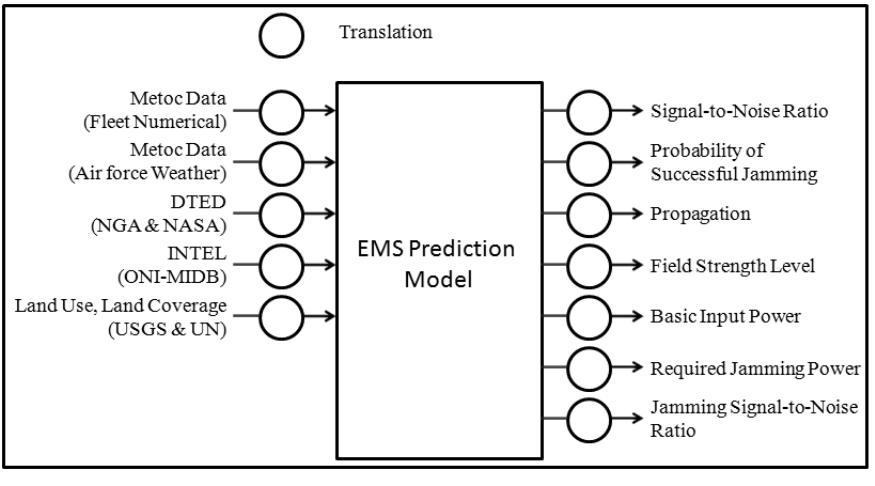


Fig. 5. EMS component and its plug-ins

6 Conclusion

In this paper we present a novel way of modeling and analyzing software compositions that advocates software reuse and increases confidence in the system composition. Although the proposed framework is limited to the types of systems that can be modeled

within the constraint logic programming schema, it can significantly impact reliability and testing processes of systems that are numerically and computationally intensive. Practicality of our approach depends on the ability to express system constraints in terms of the classes of CLP for which constraint satisfaction can be performed in an efficient time. We also describe an application of our framework within the powerful DDD software engineering methodology.

We conclude with some open issues relevant to the presented subject. We are interested in developing methods that allow us to automatically compute attributes describing inputs, outputs, and functionality of the component. In conjunction with advanced natural language techniques for system requirement processing and based on the computed component attributes we are interested in developing methods that are capable of extracting the system of constraints automatically. Component attributes should be computed based on the available documentation, hence requiring natural language processing, and based on the existing black-box (respectively gray-box) reverse engineering methods. Specifically, we are interested in computation of the domain bounds on the inputs and outputs, and collecting a sufficient number of results to verify or estimate component's functionality. Since the above computations are based on imprecise representations of somewhat uncertain information, the proposed infrastructure, needs to be integrated with validation procedures to increase the dependability of the results. Another interesting research direction is automated test scenario generators that use component attributes and functionality, and system constraints (such as domain bounds), where these compute non-trivial testing scenarios to check whether complex component realizations meet the standards associated with a given slot in an open architecture. Resolving these issues will help to reduce the gap between the conceptual verification of upgrade design and the verification of the physical implementation of the new subsystem and strengthen the chain of evidence connecting the original raw data about stakeholder needs to quality assurance procedures for concrete system components..

Acknowledgments. We would like to thank Dagohoy Anunciado from SPAWAR for helpful discussions and insightful comments.

References

1. Gen Voca, http://www.program-transformation.org/Transform/GenVoca
2. Pârv, B., Motogna, S., Lazăr, I., Czibula, I., Lazăr, L.: ComDeValCo – a framework for software component definition, validation, and composition. Studia Universitatis Babes-Bolyai Informatica LII(2), 59–68 (2007)
3. Speck, A., Pulvermüller, E., Jerger, M., Franczyk, B.: Component composition validation. International Journal of Applied Mathematics and Computer Science 12(4), 581–589 (2002)
4. Batory, D., Geraci, B.: Composition validation and subjectivity in GenVoca generators. IEEE Transactions on Software Engineering 23(2), 67–82 (1997)
5. Cicchetti, A., Di Ruscio, D.: Decoupling web application concerns through weaving operations. Sci. Comput. Program. 70(1), 62–86 (2008)
6. Lynch, N.: Distributed Algorithms. Morgan Kaufmann Publishers, San Francisco (1996)
7. Meijer, E., Drayton, P.: Static typing where possible, dynamic typing when needed (2005)
8. Choco Constraint Programming System, http://choco.sourceforge.net

9. Jaffar, J., Maher, M.: Constraint logic programming: A survey. Journal of Logic Programming 19(20), 503–581 (1994)
10. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. Prentice Hall Series in Artificial Intelligence (2003)
11. Bartak, R.: Theory and practice of constraint propagation. In: Proceedings of the 3rd Workshop on Constraint Programming in Decision and Control (2001)
12. Luqi: Dependable software architecture based on quantifiable compositional model. Technical Report NPS-CS-08-003, NPS (January 2008)
13. Deransart, P., Hermenegildo, M., Maluszynski, J. (eds.): DiSCiPl 1999. LNCS, vol. 1870. Springer, Heidelberg (2000)
14. Jaffar, J., Lassez, J.L.: Constraint logic programming. In: Proceedings of 14th ACM Symposium on Principles of Programming Languages, pp. 111–119 (1987)
15. Rossi, F., Van Beek, P., Walsh, T.: Handbook of Constraint Programming. Elsevier, Amsterdam (2006)
16. Benhamou, F., Older, W.: Programming in CLP(BNR). In: Proceedings of 1'st Workshop on Principles and Practice of Constraint Programming (1993)
17. Aiba, A., Sakai, K., Sato, Y., Hawley, D., Hasegawa, R.: Constraint logic programming language CAL. In: Proceedings of the International Conference on Fifth Generation Computer Systems, pp. 263–276 (1988)
18. Dincbas, M., Van Hentenryc, P., Simons, H., Aggoun, A.: The constraint logic programming lanugage CHIP. In: Proceedings of the 2nd International Conference on Fifth Generation Computer Systems, pp. 249–264 (1988)
19. Jaffar, J., Michaylov, S., Yap, R.: The CLP(R) language and system. ACM Transactions on Programming Languages 14(3), 339–395 (1992)
20. Colmerauer, A.: An introduction to prolog III. Communications of the ACM 33(7), 68–90 (1990)
21. Hong, H.: RISC-CLP(Real): Logic programming with non-linear constraints over the reals. In: Benhamou, F., Colmerauer, A. (eds.) Constraint Logic Programming: Selected Research, pp. 133–159. MIT Press, Cambridge (1993)
22. Fikes, R.: REF-ARF: A system for solving problems stated as procedures. 1, 27–120 (1970)
23. Delzanno, G., Podelski, A.: Constraint-based deductive model checking. International Journal on Software Tools for Technology Transfer 3(3), 250–270 (2001)
24. Van Roy, P., Haridi, S.: Concepts, Techniques, and Models of Computer Programming. MIT Press, Cambridge (2004)
25. Van Roy, P. (ed.): MOZ 2004. LNCS, vol. 3389. Springer, Heidelberg (2005) (Revised Selected and Invited Papers)
26. Luqi, L.Z., Berzins, V., Qiao, Y.: Documentation driven development for complex real-time systems. 30, 936–952 (2004)
27. Berzins, V., Rodríguez, M., Wessman, M.: Putting teeth into open architectures: Infrastructure for reducing the need for retesting, pp. 285–312 (May 2007)

Software Engineering Techniques for the Development of Systems of Systems

Radu Calinescu and Marta Kwiatkowska

Computing Laboratory, University of Oxford
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
{Radu.Calinescu,Marta.Kwiatkowska}@comlab.ox.ac.uk

Abstract. This paper investigates how existing software engineering techniques can be employed, adapted and integrated for the development of systems of systems. Starting from existing system-of-systems (SoS) studies, we identify computing paradigms and techniques that have the potential to help address the challenges associated with SoS development, and propose an SoS development framework that combines these techniques in a novel way. This framework addresses the development of a class of IT systems of systems characterised by high variability in the types of interactions between their component systems, and by relatively small numbers of such interactions. We describe how the framework supports the dynamic, automated generation of the system interfaces required to achieve these interactions, and present a case study illustrating the development of a data-centre SoS using the new framework.

1 Introduction

The functionality and flexibility underpinning today's applications in areas ranging from transportation and healthcare to aerospace and defence can no longer be provided by a monolithic information system, however complex this might be. Instead, the required capabilities can only be achieved through employing collections of collaborative, heterogeneous and autonomously-operating systems—or *systems of systems*.

The crucial importance of many systems of systems and the high rates of late delivery, over-spending and failure associated with their development have prompted the initiation of research programmes for the investigation of this new class of systems [18,24,38,41] and its extensions [15,35,42]. The results of this research provide valuable insights into the distinguishing features of systems of systems [3,5,28,36], the challenges posed by their unprecedented size, diversity, variability, complexity, unforeseen interactions and emergent behaviour [15,22,27,37], and some of the high-level principles and practices to be employed in their development [6,26,27,36].

Most importantly for the progress of the field, broad agreement has been reached on the main features that set systems of systems apart from traditional, monolithic systems:

C. Choppy and O. Sokolsky (Eds.): Monterey Workshop 2008, LNCS 6028, pp. 59–82, 2010.

- The components of a system of systems (SoS) possess a level of operational autonomy that allows them to pursue their own, local objectives, independently and in addition to contributing to the global SoS objective(s) [5,27].
- The components of a system of systems are often developed, procured and managed independently [26,36].
- SoS components may belong to multiple open and evolving systems of systems that they could join and leave dynamically [5,6,18,37].
- The behaviour of a system of systems cannot be fully predicted from the behaviour of its component systems [3,26,27,37].

Additionally, an SoS subclass that typifies key information systems of the future—i.e., the so-called *large-scale complex (IT) systems* [35] or *ultra-large-scale systems* [15,42]—is characterised by incomplete and continually changing requirements and components, and by regarding failures as normal events.

The challenges associated with the development of systems of systems are tremendous. They include the need to ensure the interoperability of a vastly diverse range of components [6,36], to convey global objectives to SoS components in meaningful ways [15,18], to achieve these objectives predictably and dependably in a dynamically changing environment [35,42], and to attain high levels of SoS longevity [15,37].

These major advances in the understanding of systems of systems laid the foundation for essential work to identify high-level principles and practices governing their development [6,15,26,27,36]. Our paper takes this work further by investigating for the first time ways in which existing software engineering techniques can contribute to the development of IT systems of systems. Thus, computer science paradigms including formal analysis and verification, model-driven and component-based development, service-oriented architectures and policy-based autonomic computing are analysed for compliance with the recommended SoS development principles, and for their ability to help address SoS engineering challenges. The results of this analysis are presented in Section 2.

In Section 3, we describe a new approach to integrating these techniques within a framework that extends the authors' previous work on quantitative analysis and verification [31,33,34], model-driven development [12] and self-* computing [7,8,10] to the realm of systems of systems. Our framework supports the development of IT systems of systems by enabling the online, automated generation of the interfaces that the system components of an SoS employ to inter-operate with the other systems within the same SoS. To achieve this, the systems to be integrated within an SoS are augmented with an *autonomic computing policy engine* that exposes their global parameters through runtime-generated interfaces defined by user-specified *policies*. This idea was originally introduced in [8], and in this paper we provide a formal description of the types of policies that can be used for this purpose and of how they can be realised in practice.

Its ability to dynamically generate new interfaces for the system components of an SoS makes our framework particularly suitable for the development of systems of systems characterised by high variability in the types of interactions between SoS component systems, and by relatively small numbers of such interactions. Therefore, the capabilities of the framework are illustrated using a case study that involves the development of a data-centre SoS with these characteristics, and which is based on a series of real-world use cases that one of the authors encountered in his previous work on a commercial system for data-centre resource management [9,11]. To add to the readability of the paper, this case study is presented as a running example spread over the next three sections.

A prototype version of the framework was implemented as an extension of our generic autonomic computing framework from [13]. Section 4 describes the novel features of this prototype implementation, and the combination of dynamic code generation, model-based development and dynamic reconfiguration techniques employed to support these features. Finally, Section 5 describes various types of IT systems of systems that can be developed using the framework, and Section 6 summarises our results and discusses a number of future research directions.

2 Software Engineering Techniques for SoS Development

This section examines existing software engineering paradigms and techniques that could help tackle some of the challenges associated with the development of systems of systems, and which are therefore likely to be part of the SoS development frameworks of the future. A summary of this analysis is presented in Table 1.

1. Service-oriented architectures (SOA). SoS development involves the integration and secure interoperation of vastly diverse technical systems [3,5,6,15,18,26,37]. Thanks to their platform independence, loose coupling and support for security, SOA solutions [46] represent strong candidates for implementing new computer systems or front-ends to legacy systems that need to be integrated into an SoS.

2. Policy-based autonomic computing. Ecosystems, cities and economies are often pointed out as examples of effective systems of systems. A common characteristic of all these systems of systems is the way in which their global objectives are specified through high-level incentives, rewards and penalties rather than by setting concrete, precise targets [15,35,36]. Thus, the behaviour of ecosystems is governed by laws of nature. The development and everyday life of cities are subject to common or civil laws and regulations. The evolution of economies is guided by taxation policies. If these successful real-world examples are to be followed, techniques will be required that can convey the global objectives of systems of systems as *high-level policies* to their autonomous components. (Policy-based) autonomic computing addresses the development of systems that

can manage themselves based on a set of high-level policies [30], and therefore represents an ideal paradigm for developing the computer-system components of an SoS.

3. Formal analysis and verification. A major concern of systems of systems is their ability to achieve an overall objective in predictable and dependable ways, through the collaboration of component systems with different (and potentially conflicting) local goals [15,35,39]. Formal analysis and verification, and in particular model checking [16], quantitative model checking [34] and quantitative analysis and verification [31,33], comprise a range of techniques that could be used or adapted for use in the verification of SoS policies, and ultimately for SoS dependability management and assurance.

4. Model-driven development and code generation. The open, evolving nature of systems of systems allows their components to join and leave dynamically [35,39]. Having SoS components collaborate with peer systems whose characteristics are often unknown until runtime is a major challenge. A combination of model-driven development and runtime code generation in which a dynamically acquired model of a peer system is used to generate the necessary interfaces and logic for collaborating with this peer system [13] represents a promising approach to addressing this challenge.

5. Component-based development. SoS engineering requires the integration of existing and future commercial, open-source and proprietary systems, and component-based development provides techniques that can help achieve this goal [1,2,17].

6. Dynamic reconfiguration. Systems of systems are required to adapt continually to changes in their environment, structure and objectives [6,26]. Recent advances in the study of dynamically reconfigurable software and hardware [19,23] provide promising approaches for the development of the computer systems to be incorporated into the systems of systems of the future.

7. Online machine learning. The levels of self-management that SoS components must achieve in impossible to anticipate circumstances are significantly beyond what can be pre-programmed into a computer system [22,35,42]. The online use of techniques specific to machine learning [4] is therefore likely to play a major role in the development of computer-based SoS components.

8. Resource discovery. In the era of mobile computing, SoS components are expected to actively seek partner systems and establish collaborations with them, thus joining (and leaving) loosely-coupled federations of systems on a regular basis [5,15,35]. The rich spectrum of resource[1] discovery techniques employed by today's distributed (e.g., grid- and web-based) computer systems [43] can be used as a basis for the development of techniques to support these capabilities.

[1] The terms *resource* and *component* are used interchangeably in the paper.

Table 1. Software engineering techniques that can help address SoS challenges

Techniques and paradigms	SoS challenges: interoperability, security	dependability (assurance)	collaboration	global-objective specification	predictability	adaptibility	longevity	flexibility
Service-oriented architectures	✓							
Policy-based autonomic computing				✓				✓
Formal analysis and verification		✓			✓			
Model-driven development/code generation						✓		
Component-based development			✓					
Dynamic reconfiguration						✓	✓	✓
Online machine learning						✓	✓	
Resource discovery			✓					✓

3 A Framework for System-of-Systems Development

3.1 Overview

Our approach to integrating the software engineering techniques analysed in the previous section involves the use of a reconfigurable policy engine with the structure in Figure 1.[2] The **SOA implementation** of this policy engine as a web service (described in Section 4) takes a model of a system and a set of policies, and ensures that the system achieves the objectives specified by these policies through adapting continually to changes in its environment.

When a running instance of the policy engine is **dynamically reconfigured** by means of a system model, its *runtime* ***code generator*** employs **model-driven development** techniques to generate *manageability adaptor proxies*, i.e., software components whose *monitor* and *control* interfaces allow the engine to read and to modify the parameters of the system components, respectively. This functionality is described in our previous work [8,13]. Additionally, the policy engine in [8,13] supports all types of policies that are standard in **policy-based autonomic computing** [44,45], and uses **resource discovery** techniques to identify the system components to which these policies need to be applied. **Component-based development** techniques introduced in Section 3.3

[2] The use of these techniques is emphasised in **bold text** in the framework overview.

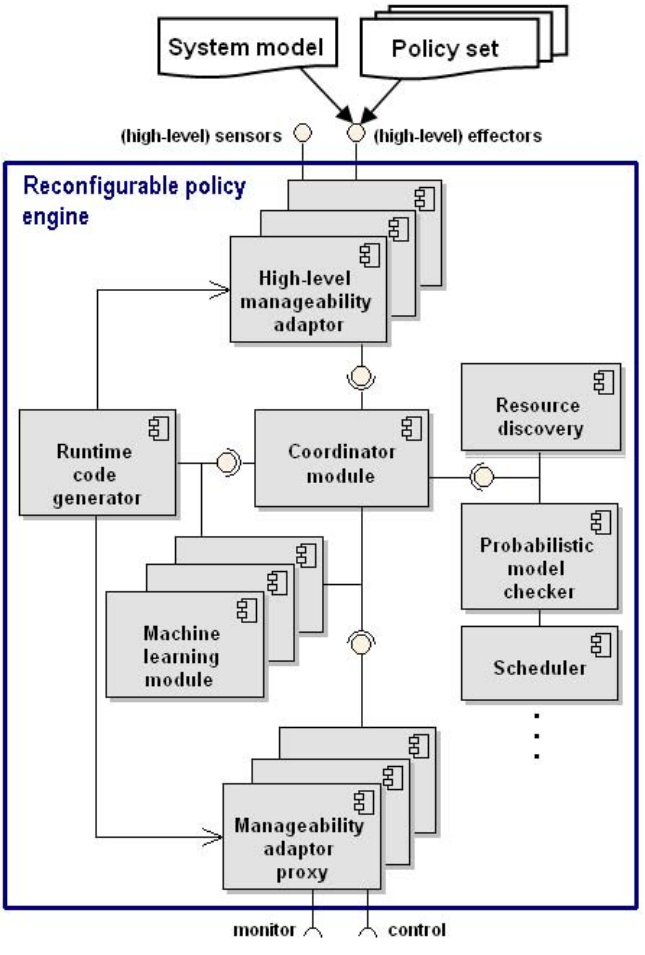


Fig. 1. Reconfigurable policy engine

are employed to integrate multiple autonomic IT systems into an autonomic system of systems.

Furthermore, we recently extended our policy engine with the ability to employ online **formal analysis and verification** techniques for the implementation of a powerful class of autonomic computing policies [10]. Finally, we are in the process of augmenting our framework with **online machine learning** capabilities by integrating it with the work presented in [20].

The remainder of this section provides brief presentations of how each software engineering technique was or, in the case of online machine learning, will be integrated into the SoS development framework. References to full descriptions of these integrations are provided for those interested in learning more about the framework.

1. Service-oriented architectures (SOA). For the reasons explained in Section 2, the components of the framework were implemented as web services. This applies to the autonomic computing policy engine from Figure 1, as well as to the software adaptors between the policy engine and the existing, legacy components that the engine is managing.

Because systems of systems comprise components that are often unknown until run time, the policy engine was provided with the capability to handle such components through run-time reconfiguration. This required the extensive use of techniques available only in an object-oriented programming environment, including reflection, polymorphism, automated generation of web-service proxies, and generic programming. Based on these requirements, J2EE and .NET were selected as candidate platforms for the framework, with .NET being eventually preferred due to its better handling of dynamic proxy generation and its slightly easier-to-use implementation of reflection.

A detailed description of the SOA implementation of the framework, and of several case studies involving the development of monolithic autonomic systems using the framework are available in [13].

2. Policy-based autonomic computing. All standard types of autonomic computing policies [29,44,45] are supported by the framework. A formal description and simple examples of these policies are provided in Section 3.2. For real-world applications of each policy type within the scope of the framework, the reader should refer to [8].

3. Formal analysis and verification. One type of autonomic computing policy supported by the framework is termed a *utility-function policy*. This policy specifies a multi-objective optimisation to be performed through continually adapting the configuration of a system to its workload and environment. Examples of utility-function policies taken from [10] are:

- optimise the parameters of a dynamically power-managed disk drive to achieve user-specified trade-offs between the response time and the power consumption of the disk drive, under variable workload;
- optimise the allocation of data-centre servers to clusters of different priorities and variable workloads, subject to using the fewest possible servers and to ensuring user-prescribed levels of cluster availability, in the presence of data-centre component failures and repairs.

To achieve such policies, the system model used to configure the policy engine includes a precise mathematical description of the system behaviour, and formally-specified quantitative properties derived from the multi-objective functions are exhaustively analysed at runtime to identify optimal system configurations. This analysis is performed using PRISM—a free, open-source tool for the formal modelling and analysis of real-time and stochastic systems [31,32] that an extensive, independent survey [25] ranked as the most effective tool for the quantitative analysis of large system models.

For the latter of the above-mentioned policies, for instance, PRISM was used to calculate the "probability that a cluster can handle its workload successfully" (i.e., the expected *cluster availability*) for each possible configuration of the cluster. This calculation was performed automatically whenever there was a significant change in the cluster workload, and the configuration that achieved the user-prescribed availability using the fewest servers was chosen.

A full description of the use of quantitative analysis within our policy engine is available from [10].

4. Model-driven development and code generation. Given that many systems of systems are characterised by dynamically changing components, the policy engine at the core of our framework was designed to handle IT components whose attributes are unknown until run time. This unique capability necessitates the run-time use of model-driven and automated code generation techniques within the policy engine. Thus, two software artefacts are generated automatically based on the system model supplied to a running instance of the policy engine: (a) data types (i.e., classes) for the new types of IT components; and (b) proxies for the adaptor web services associated with the new component types. This code generation process is discussed in detail in [13].

5. Component-based development. In this paper, we define formally a new type of autonomic computing policy that supports adaptive component-based development. Originally suggested in [8], this *resource-definition policy* specifies how the *high-level sensors* and *high-level effectors* interfaces of the policy engine should expose the system under its control as a single component, thus enabling its integration into a system of systems.

Figure 2 depicts the generic architecture of an SoS built around an extension of the policy engine from [13] that supports resource-definition policies. Each of the top-level *autonomic-enabled components* 1 to N in this architecture is a system managed by an appropriately configured instance of the policy engine. At the SoS level, the policy engine instances expose the state and configuration of their systems, employ resource discovery to identify peer SoS components, and collaborate with these. At the local level, the policy engines organise heterogeneous collections of components into a single system. These collections can comprise legacy components whose interfaces are accessed through *manageability adaptors* [13] and autonomic-enabled components (i.e., new systems that expose *sensor* and *effector* interfaces permitting their direct management by the policy engine, or other instances of the top-level autonomic-enabled components in Figure 2).

The theoretical foundation, implementation and applications of resource-definition policies are presented in Sections 3.3, 4 and 5, respectively.

6. Dynamic reconfiguration. When supplied with a new system model, the policy engine within our framework becomes capable of managing the previously unknown IT components whose characteristics are described in this model. This

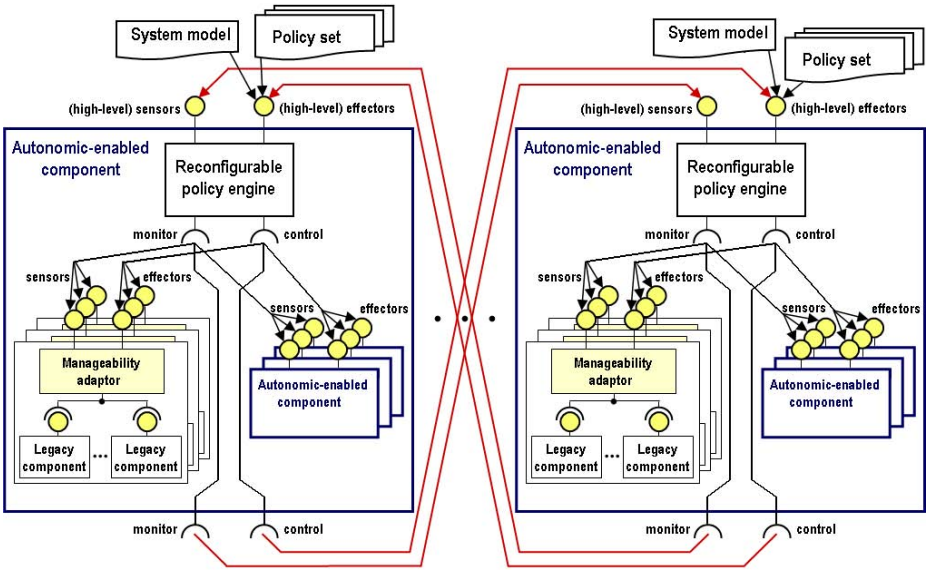


Fig. 2. System-of-systems architecture

dynamic reconfiguration of a running instance of the engine involves the automated synthesis of component-specific software artefacts for each IT component specified in the system model, as already mentioned earlier in this section and described in depth in [8,13].

7. Online machine learning. In a future version of the policy engine, online machine learning will be employed to continually improve the accuracy of the formal behavioural model that the engine uses to implement autonomic computing policies. The approach that we are working on involves learning the actual values of the model parameters from the observed behaviour of the SoS components. This extension of the framework builds on the research described in [20] and is being carried out jointly with its authors.

8. Resource discovery. The resource discovery mechanism employed by the framework has two parts. First, the framework includes a resource discovery web service that policy engine instances can query to identify the locations of the components they are required to manage—namely the URLs of the web-service adaptors that enable the engine to inter-operate with existing SoS components. Additionally, each such adaptor comprises a `SupportedResource` web method that policy engine instances call to discover the type of resource that the adaptor is exposing, as explained in detail in [8].

We will start the detailed presentation of the framework in the next section, where we formally define the system model from Figure 1 and the standard

types of autonomic computing policies supported by the policy engine. This will be followed in Section 3.3 by the specification of the resource-definition policies that underlie the component-based development of IT systems of systems using our framework.

3.2 System Model and Standard Autonomic Computing Policies

The system model used to configure the policy engine from Figure 1 is a tuple that defines the $n \geq 1$ resources of the system and their behaviour:

$$M = (R_1, R_2, \ldots, R_n, f), \tag{1}$$

where R_i, $1 \leq i \leq n$ is a formal specification for the ith system resource, and f is a model of the *known* behaviour of the system. Each resource specification R_i represents a named sequence of $m_i \geq 1$ resource parameters, i.e.,

$$R_i = (resId_i, P_{i1}, P_{i2}, \ldots, P_{im_i}), \forall 1 \leq i \leq n, \tag{2}$$

where $resId_i$ is an identifier used to distinguish between different types of resources. Finally, for each $1 \leq i \leq n$, $1 \leq j \leq m_i$, the resource parameter P_{ij} is a tuple

$$P_{ij} = (paramId_{ij}, ValueDomain_{ij}, type_{ij}) \tag{3}$$

where $paramId_{ij}$ is a `string`-valued identifier used to distinguish the different parameters of a resource; $ValueDomain_{ij}$ is the set of possible values for P_{ij}; and $type_{ij} \in \{\texttt{ReadOnly}, \texttt{ReadWrite}\}$ specifies whether the policy engine can only read or can both read and modify the value of the parameter. The parameters of each resource must have different identifiers, i.e.,

$$\forall 1 \leq i \leq n \bullet \forall 1 \leq j < k \leq m_i \bullet paramId_{ij} \neq paramId_{ik}$$

We further define the *state space* S of the system as the Cartesian product of the value domains of all its `ReadOnly` resource parameters, i.e.,

$$S = \underset{1 \leq i \leq n}{\times} \underset{\substack{1 \leq j \leq m_i \\ type_{ij} = \texttt{ReadOnly}}}{\times} ValueDomain_{ij} \tag{4}$$

Similarly, the *configuration space* C of the system is defined as the Cartesian product of the value domains of all its `ReadWrite` resource parameters, i.e.,

$$C = \underset{1 \leq i \leq n}{\times} \underset{\substack{1 \leq j \leq m_i \\ type_{ij} = \texttt{ReadWrite}}}{\times} ValueDomain_{ij} \tag{5}$$

With this notation, the behavioural model f from (1) is a partial function[3]

$$f : S \times C \nrightarrow S$$

[3] A partial function on a set X is a function whose domain is a subset of X. We use the symbol $\nrightarrow$ to denote partial functions.

such that for any $(\mathbf{s}, \mathbf{c}) \in \text{domain}(f)$, $f(\mathbf{s}, \mathbf{c})$ represents the *expected* future state of the system if its current state is $\mathbf{s} \in S$ and its configuration is set to $\mathbf{c} \in C$. Presenting classes of behavioural models that can support the implementation of different autonomic computing policies is beyond the scope of this paper; for a description of such models see [8,10].

The standard types of autonomic policies described in [29,44,45] can be defined using this notation as follows:

1. An *action policy* specifies how the system configuration should be changed when the system reaches certain state/configuration combinations:
$$p_{\text{action}} : S \times C \nrightarrow C. \tag{6}$$
Note that an action policy can be implemented even when domain(f) $= \emptyset$ in (1).
2. A *goal policy* partitions the state/configuration combinations for the system into desirable and undesirable:
$$p_{\text{goal}} : S \times C \rightarrow \{\texttt{true}, \texttt{false}\}, \tag{7}$$
with the policy engine requested to maintain the system in an operation area for which p_{goal} is `true`.
3. A *utility policy* associates a value with each state/configuration combination, and the policy engine should adjust the system configuration such as to maximise this value:
$$p_{\text{utility}} : S \times C \rightarrow \mathbb{R}. \tag{8}$$

Example 1. To illustrate the application of the notation introduced so far, consider the example of an autonomic data-centre comprising a pool of $nServers \geq 0$ servers that need to be partitioned among the $N \geq 1$ services that the data-centre can provide. Assume that each such service has a *priority* and is subjected to a variable *workload*. The model (1) for this system can be expressed as a tuple

$$M = (ServerPool, Service_1, \ldots, Service_n, f) \tag{9}$$

where the models for the server pool and for a generic service i, $1 \leq i \leq N$, are given by:

$$\begin{aligned} ServerPool &= (\texttt{"serverPool"}, \\ &\quad\ (\texttt{"nServers"}, \mathbb{N}, \texttt{ReadOnly}), \\ &\quad\ (\texttt{"partition"}, \mathbb{N}^N, \texttt{ReadWrite})) \\ Service_i &= (\texttt{"service"}, \\ &\quad\ (\texttt{"priority"}, \mathbb{N}_+, \texttt{ReadOnly}), \\ &\quad\ (\texttt{"workload"}, \mathbb{N}, \texttt{ReadOnly})) \end{aligned} \tag{10}$$

The state and configuration spaces of the system are $S = \mathbb{N} \times (\mathbb{N}_+ \times \mathbb{N})^N$ and $C = \mathbb{N}^N$, respectively. For simplicity, we will consider that the *workload* of a service represents the minimum number of operational servers it requires to achieve

its service-level agreement. Sample action, goal and utility policies for the system are specified below by giving their values for a generic data-centre state $\mathbf{s} = (n, p_1, w_1, p_2, w_2, \ldots, p_N, w_N) \in S$ and configuration $\mathbf{c} = (n_1, n_2, \ldots, n_N) \in C$:

$$p_{\text{action}}(\mathbf{s}, \mathbf{c}) = (\lceil \alpha w_1 \rceil, \lceil \alpha w_2 \rceil, \ldots, \lceil \alpha w_N \rceil) \tag{11}$$

$$p_{\text{goal}}(\mathbf{s}, \mathbf{c}) = \forall 1 \leq i \leq N \bullet (n_i > 0 \implies (\forall 1 \leq j \leq N \bullet p_j > p_i \implies n_j = \lceil \alpha w_j \rceil)) \tag{12}$$

$$p_{\text{utility}}(\mathbf{s}, \mathbf{c}) = \begin{cases} -\infty, & \text{if } \sum_{i=1}^{N} n_i > n \\ \sum_{\substack{i=1 \\ w_i > 0}}^{N} p_i u(w_i, n_i) - \epsilon \sum_{i=1}^{N} n_i, & \text{otherwise} \end{cases} \tag{13}$$

We will describe each of these policies in turn. First, the action policy (11) prescribes that $\lceil \alpha w_i \rceil$ servers are allocated to service i, $1 \leq i \leq N$, at all times. Notice how a *redundancy factor* $\alpha \in (1, 2)$ is used in a deliberately simplistic attempt to increase the likelihood that at least w_i servers will be available for service i in the presence of server failures. Also, the policy is (over)optimistically assuming that $n \geq \sum_{i=1}^{N} \lceil \alpha w_i \rceil$ at all times.

The goal policy (12) specifies that the desirable state/configuration combinations of the data-centre are those in which service i, $1 \leq i \leq N$, is allocated servers only if all services of higher priority have already been allocated $\lceil \alpha w_i \rceil$ servers.

Finally, the utility policy requires that the value of the expression in (13) is maximised. The value $-\infty$ in this expression is used to avoid the configurations in which more servers than the n available are allocated to the services. When this is not the case, the value of the policy is given by the combination of two sums. The first sum encodes the utility $u(w_i, n_i)$ of allocating n_i servers to each service i with $w_i > 0$, weighted by the priority p_i of the service. By setting ϵ to a small positive value (i.e., $0 < \epsilon \ll 1$), the second sum ensures that from all server partitions that maximise the first sum, the one that uses the smallest number of

$$u : \mathsf{R}_+ \times \mathsf{R}_+ \to [0, 1]$$

$$u(w, n) = \begin{cases} 0, & \text{if } n < (2 - \alpha)w \\ \frac{n - (2-\alpha)w}{2(\alpha - 1)w}, & \text{if } (2 - \alpha)w \leq n \leq \alpha w \\ 1, & \text{if } n > \alpha w \end{cases}$$

$$\alpha \in (1, 2)$$

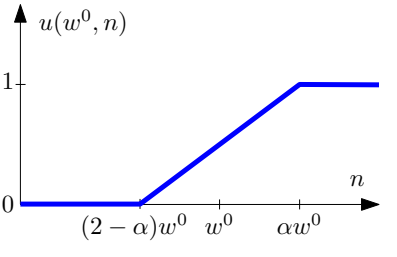


Fig. 3. Sample function u for Example 1 (the graph shows u for a fixed value w^0 of its first argument)

servers is chosen at all times. A sample function u is shown in Figure 3; a more realistic u and a matching behavioural model f from (9) are described in [8].

3.3 Resource-Definition Policies for Runtime Interface Generation

Using $\mathcal{R}$ to denote the set of all resource models with the form in (2), and $\mathcal{E}(S, C)$ to denote the set of all expressions defined on the Cartesian product $S \times C$, we can now give the generic form of a resource-definition policy as

$$p_{\text{def}} : S \times C \to \mathcal{R} \times \mathcal{E}(S, C)^q, \tag{14}$$

where, for any $(\mathbf{s}, \mathbf{c}) \in S \times C$,

$$p_{\text{def}}(\mathbf{s}, \mathbf{c}) = (R, E_1, E_2, \ldots, E_q). \tag{15}$$

In this definition, R represents the resource that the policy engine is required to synthesise, and the expressions E_1, E_2, ..., E_q specify how the engine will calculate the values of the $q \geq 0$ `ReadOnly` parameters of R as functions of $(\mathbf{s}, \mathbf{c})$. Assuming that $q > 0$ and the value domain for the ith `ReadOnly` parameter of R, $1 \leq i \leq q$ is $\mathit{ValueDomain}_i$, we have $E_i : S \times C \to \mathit{ValueDomain}_i$.

Example 2. Consider again the autonomic data-centre from Example 1. A sample resource-definition policy that complements the utility policy in (13) is given by

$$\begin{aligned} p_{\text{def}}(s, c) = ((&\texttt{"dataCentre"}, \\ &(\texttt{"id"}, \mathit{String}, \texttt{ReadOnly}) \\ &(\texttt{"maxUtility"}, \mathbb{R}, \texttt{ReadOnly}), \\ &(\texttt{"utility"}, \mathbb{R}, \texttt{ReadOnly})), \\ &\texttt{"dataCentre A"}, \\ &\max_{(x_1, x_2, \ldots, x_N) \in \mathbb{N}^N} \textstyle\sum_{w_i > 0}^{1 \leq i \leq N} p_i u(w_i, x_i), \\ &\textstyle\sum_{w_i > 0}^{1 \leq i \leq N} p_i u(w_i, n_i)) \end{aligned} \tag{16}$$

This policy requests the synthesis of a resource termed a `"dataCentre"`. This resource comprises three `ReadOnly` parameters: `id` is a string-valued identifier with the constant value `"dataCentre A"`, while `maxUtility` and `utility` represent the maximum and actual utility values associated with the autonomic data-centre when it implements the utility policy (13). (The term $\epsilon \sum_{i=1}^{N} n_i$ from the policy definition is insignificant, and was not included in (16) for simplicity.) Exposing the system through this synthesised resource enables an external policy engine to monitor how close the data-centre is to achieving its maximum utility.

Note that the generic form of a resource-definition policy (14)-(15) allows users to request the policy engine to synthesise different types of resources for different state/configuration combinations of the system. While the preliminary use cases that we have studied so far can be handled using resource-definition policies in which the resource model R from (15) is fixed for all $(\mathbf{s}, \mathbf{c}) \in S \times C$, we envisage

that this capability will be useful for more complex applications of resource-definition policies.

We will next explore the semantics and applications of `ReadWrite` (i.e., configurable) parameters in synthesised resources. These are parameters whose identifiers and value domains are specified through a resource-definition policy, but whose values are set by an external entity such as another policy engine. Because these parameters do not correspond to any element of the managed resources within the autonomic system, the only way ensure that they have an influence on an individual system from the SoS architecture in Figure 2 is to take them into account within the set of policies implemented by the policy engine associated with that system. This is achieved by redefining the state space S of the system. Thus, in the presence of resource-definition policies requesting the synthesis of high-level resources with a non-empty set of `ReadWrite` parameters $\{P_1^{\text{synth}}, P_2^{\text{synth}}, \ldots, P_r^{\text{synth}}\}$, the state space definition (4) is replaced by:

$$S = \left(\underset{1 \le i \le n}{\times} \underset{\substack{1 \le j \le m_i \\ type_{ij} = \texttt{ReadOnly}}}{\times} ValueDomain_{ij} \right) \times \left(\underset{1 \le i \le r}{\times} ValueDomain_i^{\text{synth}} \right), \tag{17}$$

where $ValueDomain_i^{\text{synth}}$ represents the value domain of the ith synthesised resource parameter P_i^{synth}, $1 \le i \le r$.

Example 3. Consider again our running example of an autonomic data-centre. The resource-definition policy in (16) can be extended to allow a peer data-centre (such as a data-centre running the same set of services within the same security domain) to take advantage of any spare servers:

$$\begin{aligned} p'_{\text{def}}(s,c) = ((&\texttt{"dataCentre"}, \\ &(\texttt{"id"}, \mathit{String}, \texttt{ReadOnly}) \\ &(\texttt{"maxUtility"}, \mathbb{R}, \texttt{ReadOnly}), \\ &(\texttt{"utility"}, \mathbb{R}, \texttt{ReadOnly})), \\ &(\texttt{"nSpare"}, \mathbb{N}, \texttt{ReadOnly})), \\ &(\texttt{"peerRequest"}, \mathbb{N}^N, \texttt{ReadWrite})), \\ &\texttt{"dataCentre A"}, \\ &\max_{(x_1, x_2, \ldots, x_N) \in \mathbb{N}^N} \textstyle\sum_{w_i > 0}^{1 \le i \le N} p_i u(w_i, x_i), \\ &\textstyle\sum_{w_i > 0}^{1 \le i \le N} p_i u(w_i, n_i), \\ &n - \textstyle\sum_{i=1}^{N} n_i) \end{aligned} \tag{18}$$

The synthesised resource has two new parameters: `nSpare` represents the number of servers not allocated to any (local) service; and `peerRequest` is a vector $(n_1^l, n_2^l, \ldots, n_N^l)$ that a remote data-centre can set to request that the local data-centre assigns n_i^l of its servers to service i, for all $1 \le i \le N$.

To illustrate how this is achieved, we will consider two data-centres that each implements the policy in (18), and which have access to each other's "dataCentre" resource as shown in the lower half of Figure 5 from one of the next sections of the paper. For simplicity, we will further assume that the data-centres are responsible for disjoint sets of services (i.e., there is no $1 \leq i \leq N$ such that $w_i > 0$ for both data-centres). To ensure that the two data-centres collaborate, we need policies that specify how each of them should set the $\texttt{peerRequest}^r$ parameter of its peer, and how it should use its own $\texttt{peerRequest}^l$ parameter (which is set by the other data-centre). The "dataCentre" parameters have been annotated with the superscripts ^{l} and ^{r} to distinguish between identically named parameters belonging to the *local* and *remote* data-centre, respectively. Before giving a utility policy that ensures the collaboration of the two data-centres, it is worth mentioning that the state of each has the form $\mathbf{s} = (n, p_1, w_1, p_2, w_2, \ldots, p_N, w_N, n^r, n_1^l, n_2^l, \ldots, n_N^l)$ (cf. (17)); and the system configuration has the form $\mathbf{c} = (n_1, n_2, \ldots, n_N, n_1^r, n_2^r, \ldots, n_N^r)$. The utility policy to use alongside policy (18) is given below:

$$p'_{\text{utility}}(\mathbf{s}, \mathbf{c}) = \begin{cases} -\infty, & \text{if } \sum_{i=1}^{N} n_i > n \vee \sum_{i=1}^{N} n_i^r > n^r \\ \sum\limits_{\substack{i=1 \\ w_i > 0}}^{N} p_i u(w_i, n_i + n_i^r) - \epsilon \sum\limits_{i=1}^{N} n_i - & \\ \quad -\lambda \sum\limits_{i=1}^{N} n_i^r + \mu \sum\limits_{\substack{i=1 \\ n_i^l > 0}}^{N} \min\left(1, \frac{n_i}{n_i^l}\right) & \text{, otherwise} \end{cases} \tag{19}$$

where $0 < \epsilon \ll \lambda, \mu \ll 1$ are user-specified constants. The value $-\infty$ is used to avoid the configurations in which more servers than available (either locally or from the remote data-centre) are allocated to the local services. The first two sums in the expression that handles all other scenarios are similar to those from utility policy (13), except that $n_i + n_i^r$ rather than n_i servers are being allocated to any local service i for which $w_i > 0$. The term $-\lambda \sum_{i=1}^{N} n_i^r$ ensures that the optimal utility is achieved with as few remote servers as possible, and the term $\mu \sum_{n_i^l > 0}^{1 \leq i \leq N} \min(1, \frac{n_i}{n_i^l})$ requests the policy engine to allocate local servers to services for which $n_i^l > 0$. Observe that the contribution of a term $\mu \min(1, \frac{n_i}{n_i^l})$ to the overall utility increases as n_i grows from 0 to n_i^l, and stays constant if n_i increases beyond n_i^l. Together with the utility term $-\epsilon \sum_{i=1}^{N} n_i$, this determines the policy engine to never allocate more than the requested n_i^l servers to service i. Small positive constants are used for the weights ϵ, λ and μ so that the terms they belong to are negligible compared to the first utility term. Further, choosing $\epsilon \ll \lambda$ ensures that using a local server decreases the utility less than using a remote one; and setting $\epsilon \ll \mu$ ensures that allocating up to n_i^l servers to a service i at the request of the remote data-centre increases the system utility.

Finally, note that because the requests for remote servers and the allocation of such servers take place asynchronously, there is a risk that the parameter

values used in policy (19) may be out of date.[4] However, this is not a problem, as the allocation of fewer or more remote servers than ideally required is never decreasing the utility value for a data-centre below the value achieved when the data-centre operates in isolation. Additionally, local servers are never used for remote services at the expense of the local services because $\sum_{w_i>0}^{1\leq i\leq N} p_i u(w_i, n_i) \gg \mu \sum_{n_i^l>0}^{1\leq i\leq N} \min(1, n_i/n_i^l))$ in the utility expression.

4 Prototype Implementation

The *policy engine* introduced in [13] was extended with the ability to handle the new type of autonomic computing policy. Implemented as a model-driven, service-oriented architecture with the characteristics presented in [12], the policy engine from [13] can manage IT resources whose model is supplied to the engine in a runtime configuration step. The IT resource models are represented as XML documents that are instances of a pre-defined meta-model encoded as an XML schema [12,13]. This choice was motivated by the availability of numerous off-the-shelf tools for the manipulation of XML documents and XML schemas—a characteristic largely lacking for the other technologies we considered. The policy engine is implemented as a .NET web service, and takes advantage of object-oriented technology features such as polymorphism, reflection[5] and generics[6] in its handling of IT resources whose characteristics are unknown until runtime.

The manageability adaptors from Figure 2 are implemented by the framework in [13] as web services that specialise a generic, abstract web service *ManagedResource*$\langle\rangle$. For each type of resource in the system, a manageability adaptor is built in two steps. First, a class (i.e., a data type) T_i is generated from the resource model (2) that will be used to configure the policy engine. Second, the manageability adaptor $\mathsf{ManagedT}_i$ for resources of type T_i is implemented by specialising our generic *ManagedResource*$\langle\rangle$ adaptor, i.e., $\mathsf{ManagedT}_i$: *ManagedResource*$\langle \mathsf{T}_i \rangle$. This process is described in [13].

Adding support for the implementation of the resource-definition policy in (14)–(15) involved extending the policy engine described above with the following functionality:

1. Automated generation of a .NET class T for the synthesised resource R from (15). This class is built by including a field and the associated getter/setter methods for each parameter of R. The types of these fields are given by the value domains of the resource parameters.

[4] In practical scenarios that we investigated this happened very infrequently relative to the time required to solve the linear optimisation problem (19) automatically within the policy engine.

[5] Reflection is an object-oriented programming technique that allows the runtime discovery and creation of objects based on their metadata [40].

[6] Generics or generic programming represents an object-oriented programming technique enabling code to be written in terms of data types unknown until runtime [21].

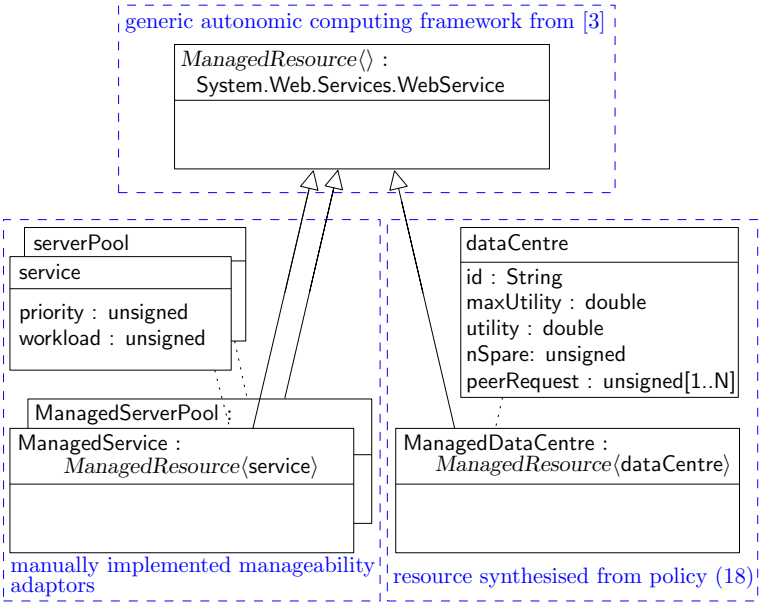


Fig. 4. Class diagram for Example 4

2. Automated creation of an instance of T. Reflection is employed to create an instance of T for the lifespan of the resource-definition policy. The `ReadOnly` fields of this object are updated by the policy engine using the expressions $E_1, E_2, \ldots, E_q$ whenever the object is accessed by an external entity.
3. Automatic generation of a manageability adaptor web service ManagedT : *ManagedResource*⟨T⟩. The web methods provided by this manageability adaptor allow entities from outside the autonomic system (e.g., external policy engines) to access the object of type T maintained by the policy engine. The fields of this object that correspond to `ReadOnly` parameters of R can be read, and those corresponding to `ReadWrite` parameters can be read and modified, respectively.

The .NET components generated in steps 1 and 3 are deployed automatically, and made accessible through the same Microsoft IIS instance as the policy engine. The synthesised IT resource is available as soon as the engine completes its handling of the resource-definition policy.

Example 4. Returning to our running example of an autonomic data-centre, the class diagram in Figure 4 depicts the manageability adaptors in place after policy (18) was supplied to the policy engine. Thus, the ManagedServerPool and ManagedService classes in this diagram represent the manageability adaptors implemented manually for the *ServerPool* and *Service* resources described in Example 1. The other manageability adaptor derived from *ManagedResource*⟨⟩

(i.e., ManagedDataCentre) was synthesised automatically by the policy engine as a result of handling the resource-definition policy.

Also shown in the diagram are the classes used to represent instances of the IT resources within the system—serverPool and service for the original autonomic system, and dataCentre for the resource synthesised from policy (18). Notice the one-to-one mapping between the fields of these classes and the parameters of their associated resources (described in Examples 1 and 3).

5 System of Systems Development

System-of-systems application development using the framework described in Sections 3 and 4 involves supplying resource-definition policies to existing autonomic systems whose policy engines support the new policy type. Hierarchical systems of systems can then be built by setting a higher-level policy engine to monitor and/or control the resources synthesised as a result of implementing these policies. Alternatively, the original autonomic systems can be configured to collaborate with each other by means of the synthesised resource sensors and effectors. Hybrid applications comprising both types of interactions mentioned above are also possible, as illustrated by the following example.

Example 5. The policy engine from Section 4 was used to simulate an autonomic system of systems comprising the pair of autonomic data-centres described in Example 3, and a top-level policy engine that monitors and summarises their performance using a dashboard resource (Figure 5). The policies implemented

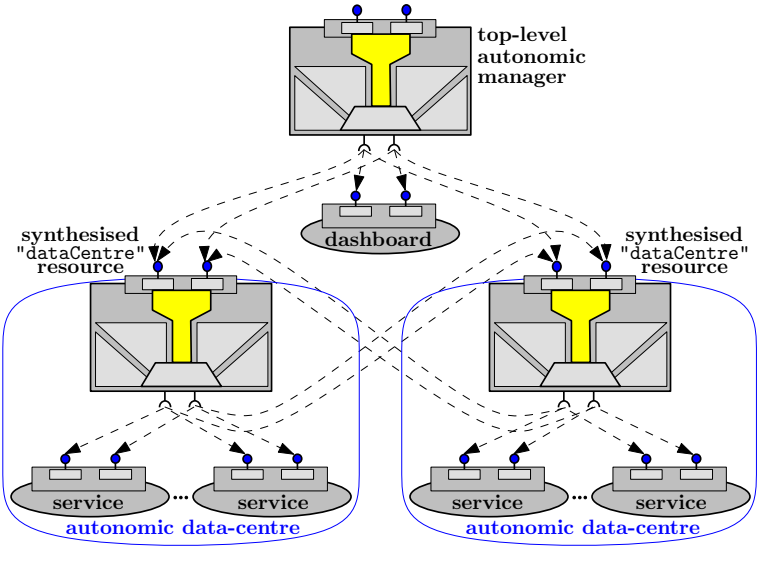


Fig. 5. Autonomic system of systems for Example 5

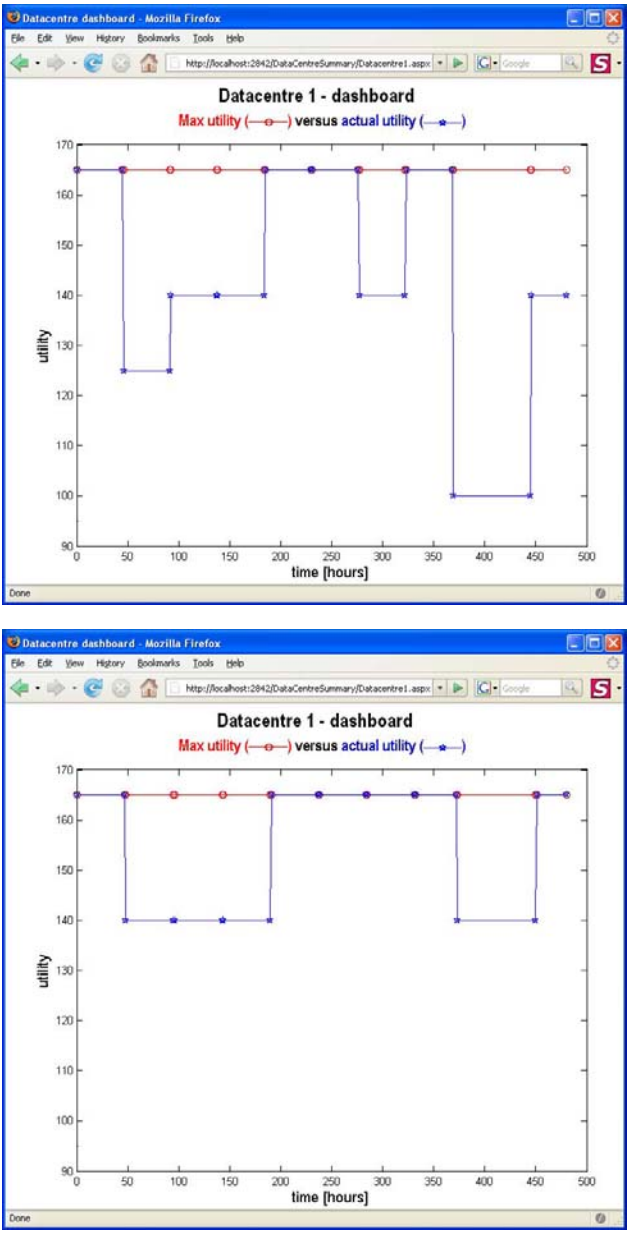


Fig. 6. Dashboard for isolated data-centre (top) and for identical data-centre operating as part of the autonomic system of systems from Figure 5 (bottom)

by the policy engines local to each data-centre are policies (18)–(19) from Example 3. The top-level policy engine implements a simple action policy that periodically copies the values of the `maxUtility` and `utility` parameters of the `"dataCentre"` resources synthesised by the data-centres into the appropriate `ReadWrite` parameters of the dashboard. For brevity, we do not give this policy here; a sample action policy was presented earlier in Example 1.

We used the data-centre resource simulators from [8], and implemented the dashboard resource as an ASP.NET web page provided with a manageability adaptor built manually as described in Section 4 and in [13]. Separate series of experiments for 20-day simulated time periods were run for two scenarios. In the first scenario, the data-centres were kept operating in isolation, by blocking the mechanisms they could use to discover each other. In the second scenario, the data-centres were allowed to discover each other, and thus to collaborate through implementing policy (19). Figure 6 depicts typical snapshots of the dashboard for both scenarios and for one of the data-centres; the same simulated service workloads were used in both experiments shown. As expected from the analysis in Example 3, the system achieves higher utility when data-centre collaboration is enabled, thus allowing data-centres to utilise each other's spare servers.

6 Conclusion

A common finding of SoS studies is that existing techniques and tools are unable to address the whole spectrum of challenges associated with the development of systems of systems [3,15,22,27]. Notwithstanding the disparity between what can be achieved using current approaches and these challenges, the SoS development frameworks of the future are likely to incorporate some of today's software engineering techniques or adapted, enhanced variants of them. This paper examined techniques that are candidates for this role, including formal analysis and verification, model-driven development, service-oriented architectures, component-based development and policy-based autonomic computing. Having first identified the SoS challenge(s) that each such technique can help address, we then proposed a new approach to combining these techniques into a framework for the development of a class of IT systems of systems.

The administrators of an SoS developed using our framework can specify at run time how the SoS components inter-operate with each other. To handle runtime changes in this specification, our framework employs a combination of model-based and online code generation techniques to automatically build and deploy the interfaces necessary to support new types of inter-operation among SoS components. This capability is particularly useful given that SoS components often belong to open, evolving systems of systems that they could join and leave dynamically [5,6,18,37]. Additionally, our framework is suitable for the development of IT systems of systems that need to adapt their inter-component interactions to changes in the SoS global objectives and/or context.

The automation of time-demanding processes such as the run-time synthesis of the interfaces between SoS component systems and the reconfiguration of the

policy engine represents a key benefit of the framework. The use of an earlier version of the framework to develop monolithic autonomic systems yielded a tenfold reduction in development time [14], and preliminary experiments with its extended prototype presented in Section 4 indicate that significant reductions are also possible in the case of systems of systems.

In the current version of our SoS development framework, the dynamically generated interfaces among SoS component systems can be used only in asynchronous mode, and involve periodical polling by the reconfigurable policy engines within the SoS. For this reason, the systems of systems whose development is currently supported by the framework are those characterised by asynchronous and relatively infrequent interactions between SoS component systems. A temporary workaround for this limitation is to mix dynamically generated component interfaces suffering from this constraint with statically implemented interfaces. For a long-term solution, we are investigating the possibility to use a notification mechanism within our reconfigurable policy engine in order to support the runtime specification of synchronous SoS component interfaces.

Additional areas of future work include the validation of the proposed framework within new application domains, the development of SoS-specific online machine learning techniques, the synthesis of high-level SoS policies from specifications, and the design of metrics for the assessment of global SoS effectiveness.

Acknowledgement

This work was partly supported by the UK Engineering and Physical Sciences Research Council Grant EP/F001096/1.

References

1. Allen, R., Garlan, D.: A formal basis for architectural connection. ACM Transactions on Software Engineering and Methodology 6(3), 213–249 (1997)
2. Arbab, F.: Abstract behavior types: a foundation model for components and their composition. Science of Computer Programming 55(1-3), 3–52 (2005)
3. Bar-Yam, Y., et al.: The characteristics and emerging behaviors of system-of-systems. Complex physical, biological and social systems project report, New England Complex Systems Institute (January 2004), http://necsi.org/education/oneweek/winter05/NECSISoS.pdf
4. Bishop, C.M.: Pattern Recognition and Machine Learning. Springer, Heidelberg (2007)
5. Boardman, J., Sauser, B.: System of systems – the meaning of of. In: Proceedings of the 2006 IEEE/SMC International Conference on System of Systems Engineering, pp. 118–123 (2006)
6. Brownsword, L., Fisher, D., Morris, E., Smith, J., Kirwan, P.: System-of-systems navigator: An approach for managing system-of-systems interoperability. Technical Report CMU/SEI-2006-TN-019, Carnegie Mellon Software Engineering Institute (April 2006), http://www.sei.cmu.edu/pub/documents/06.reports/pdf/06tn019.pdf

7. Calinescu, R.: Towards a generic autonomic architecture for legacy resource management. In: Elleithy, K. (ed.) Innovations and Advanced Techniques in Systems, Computing Sciences and Software Engineering, pp. 410–415. Springer, Heidelberg (2008)
8. Calinescu, R.: General-purpose autonomic computing. In: Denko, M., et al. (eds.) Autonomic Computing and Networking, pp. 3–20. Springer, Heidelberg (2009)
9. Calinescu, R., Hill, J.M.D.: System providing methodology for policy-based resource allocation. US Patent Application 20050149940 (July 2005)
10. Calinescu, R., Kwiatkowska, M.: Using quantitative analysis to implement autonomic IT systems. In: Proceedings of the 31st International Conference on Software Engineering (ICSE 2009) (May 2009)
11. Calinescu, R.: Challenges and best practices in policy-based autonomic architectures. In: Proceedings of the 3rd IEEE International Symposium on Dependable, Autonomic and Secure Computing (DASC 2007), Columbia, Maryland, USA, pp. 65–74 (2007)
12. Calinescu, R.: Model-driven autonomic architecture. In: Proceedings of the 4th IEEE International Conference on Autonomic Computing, Jacksonville, Florida (June 2007)
13. Calinescu, R.: Implementation of a generic autonomic framework. In: Greenwood, D., et al. (eds.) Proceedings 4th International Conference on Autonomic and Autonomous Systems (ICAS 2008), pp. 124–129. IEEE Computer Society Press, Los Alamitos (2008)
14. Calinescu, R., Kwiatkowska, M.: CADS*: Computer-aided development of self-* systems. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 421–424. Springer, Heidelberg (2009)
15. Carnegie Mellon Software Engineering Institute. Ultra-Large-Scale Systems. The Software Challenge of the Future. Carnegie Mellon University (2006), http://www.sei.cmu.edu/uls/files/ULS_Book2006.pdf
16. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (2000)
17. Crnkovic, I., Larsson, M. (eds.): Building Reliable Component-Based Software Systems. Artech House Publishers (2002)
18. Crossley, W.A.: System of Sytems: An Introduction of Purdue University Schools of Engineering's Signature Area. In: Proceedings of the Engineering Systems Symposium (2004), http://esd.mit.edu/symposium/pdfs/papers/crossley.pdf
19. El-Ghazawi, T., El-Araby, E., Huang, M., Gaj, K., Kindratenko, V., Buell, D.: The promise of high-performance reconfigurable computing. Computer 41(2), 69–76 (2008)
20. Epifani, I., Ghezzi, C., Mirandola, R., Tamburrelli, G.: Model evolution by run-time adaptation. In: Proceedings of the 31st International Conference on Software Engineering (ICSE 2009), May 2009, pp. 111–121 (2009)
21. Garcia, R., et al.: A comparative study of language support for generic programming. ACM SIGPLAN Notices 38(11), 115–134 (2003)
22. Goth, G.: Ultralarge systems: Redefining software engineering? IEEE Software 25(3), 91–94 (2008)
23. Hannebauer, M.: Autonomous Dynamic Reconfiguration in Multi-agent Systems. Springer, Heidelberg (2002)
24. Integration of Software-Intensive Systems (ISIS) Initiative: Addressing System-of-Systems Interoperability, http://www.sei.cmu.edu/isis

25. Jansen, D.N., et al.: How fast and fat is your probabilistic model checker? An experimental comparison. In: Yorav, K. (ed.) HVC 2007. LNCS, vol. 4899, pp. 69–85. Springer, Heidelberg (2008)
26. Kaplan, J.M.: A new conceptual framework for net-centric, enterprise-wide, system-of-systems engineering. Defense & Technology Papers 30, US Center for Technology and National Security Policy (July 2006), http://www.ndu.edu/ctnsp/Def_Tech/DTP%2030%20A%20New%20Conceptual%20Framework.pdf
27. Keating, C.: Research foundations for system of systems engineering. In: 2005 IEEE International Conference on Systems, Man and Cybernetics, vol. 3, pp. 2720–2725 (2005)
28. Keating, C., Rogers, R., Unal, R., Dryer, D., Sousa-Poza, A., Safford, R., Peterson, W., Rabadi, G.: System of systems engineering. Engineering Management Journal 15(3), 36–45 (2003)
29. Kephart, J.O., Walsh, W.E.: An artificial intelligence perspective on autonomic computing policies. In: Proc. 5th IEEE Intl. Workshop on Policies for Distributed Systems and Networks (2004)
30. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Computer Journal 36(1), 41–50 (2003)
31. Kwiatkowska, M., Norman, G., Parker, D.: Probabilistic model checking in practice: Case studies with PRISM. ACM SIGMETRICS Performance Evaluation Review 32(4), 16–21 (2005)
32. Kwiatkowska, M., Norman, G., Parker, D.: Quantitative analysis with the probabilistic model checker PRISM. Electronic Notes in Theoretical Computer Science 153(2), 5–31 (2005)
33. Kwiatkowska, M.: Quantitative verification: Models, techniques and tools. In: Proc. 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), September 2007, pp. 449–458. ACM Press, New York (2007)
34. Kwiatkowska, M., Norman, G., Parker, D.: Stochastic model checking. In: Bernardo, M., Hillston, J. (eds.) SFM 2007. LNCS, vol. 4486, pp. 220–270. Springer, Heidelberg (2007)
35. LSCITS Consortium. Large-Scale Complex Information Technology Systems Initiative, http://www.lscits.org
36. Maier, M.W.: Architecting principles for systems-of-systems. Systems Engineering 1(4), 267–284 (1999)
37. Meilich, A.: System of systems engineering (SoSE) and architecture challenges in a net centric environment. In: Proceedings of the 2006 IEEE/SMC International Conference on System of Systems Engineering, pp. 1–5 (2006)
38. US National Centers for Systems of Systems Engineering (NCSOSE), http://www.eng.odu.edu/ncsose/
39. Popper, S.W., Bankes, S.C., Callaway, R., DeLaurentis, D.: System of systems symposium: Report on a summer conversation. In: Proceedings of the 1st System of Systems Symposium (2004), http://www.potomacinstitute.org/academiccen/SoS%20Summer%20Conversation%20report.pdf
40. Sobel, J.M., Friedman, D.P.: An introduction to reflection-oriented programming. In: Proceedings of Reflection 1996 (1996)
41. US System of Systems Engineering Center of Excellence (SoSECE), http://www.sosece.org

42. US Industry/University Collaborative Research Center for Ultra-Large-Scale Software-Intensive Systems (ULSSIS), http://ulssis.cs.virginia.edu
43. Vanthournout, K., Deconinck, G., Belmans, R.: A taxonomy for resource discovery. Personal and Ubiquitous Computing 9(2), 81–89 (2005)
44. Walsh, W.E., et al.: Utility functions in autonomic systems. In: Proc. 1st Intl. Conf. Autonomic Computing, pp. 70–77 (2004)
45. White, S.R., et al.: An architectural approach to autonomic computing. In: Proc. 1st IEEE Intl. Conf. Autonomic Computing, pp. 2–9. IEEE Computer Society, Los Alamitos (2004)
46. Zimmermann, O., et al.: Perspectives on Web Services: Applying SOAP, WSDL and UDDI to Real-World Projects. Springer, Heidelberg (2005)

Simulation of LET Models in Simulink and Ptolemy

Patricia Derler, Andreas Naderlinger, Wolfgang Pree, Stefan Resmerita, and Josef Templ

C. Doppler Laboratory Embedded Software Systems
University of Salzburg
firstname.lastname@cs.uni-salzburg.at

Abstract. This paper describes two different approaches of simulating embedded control software whose real-time requirements are explicitly specified by means of the Logical Execution Time (LET) abstraction introduced in the Giotto project. As simulation environments we chose the black-box MATLAB/Simulink product and the open-source project Ptolemy II. The paper first sketches the modeling of LET-based components with the Timing Definition Language (TDL). As the LET abstraction allows the platform-independent modeling of the timing behavior of embedded software, a correct simulation of TDL components is equivalent to the behavior on a specific platform. We integrated TDL with both MATLAB/Simulink and Ptolemy and highlight the differences and similarities of the particular TDL simulation.

Keywords: simulation of real-time behavior, real-time modeling, Simulink, Ptolemy, Logical Execution Time (LET), Timing Definition Language (TDL).

1 Introduction

This work compares the way simulation environments with different goals facilitate the simulation of embedded control software modeled with the Timing Definition Language (TDL) [2]. TDL harnesses the Logical Execution Time (LET) abstraction for deterministically describing the timing behavior of a set of periodic tasks. If the tasks can be scheduled for a specific, potentially distributed platform given the worst-case execution time for each task for each computing node, the observable behavior of the system will be the same on the platform as in a simulation given the same inputs.

MATLAB/Simulink is a commercially available tool suite used to simulate control systems and also to generate C code. Simulink defines a fixed model of computation (MoC) that can only be adapted to some extent by means of so-called solvers as well as via the triggering of block executions. Ptolemy is an open-source simulation environment that serves as playground for experimenting with different MoCs and their combination in heterogeneous models.

We implemented the TDL model of computation in both, Simulink and Ptolemy, and describe the different integration approaches we had to take. We first introduce TDL.

C. Choppy and O. Sokolsky (Eds.): Monterey Workshop 2008, LNCS 6028, pp. 83–92, 2010.

Sections 2 and 3 sketch the core concepts how TDL components are modeled and simulated in Simulink and Ptolemy. Section 4 compares the two approaches.

Timing Definition Language (TDL). While TDL [2] is conceptually based on the LET-abstraction introduced in the Giotto project [3], it provides extended features, a more convenient syntax, and an improved set of programming tools. The LET associated with a computational unit, called task, represents the duration between the time instant when the task becomes ready for execution and the instant when the task terminates. A task's LET is specified independently of the task's functionality. When deploying the model on a platform, the LET specification is satisfied if the total physical execution time of the task is included in the LET interval for every task invocation, and an appropriate runtime system ensures that task inputs are read at the beginning of the LET interval (the release time) and task outputs are made available at the end of the LET interval (the termination time). Figure 1 illustrates the LET abstraction for one task invocation. Between release and termination points, the output values are those established in the previous execution; default or specified initial values are used during the first execution.

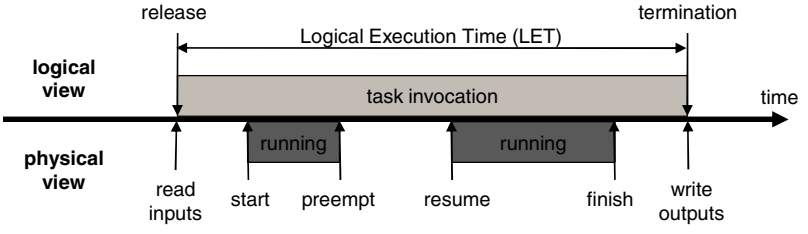


Fig. 1. Logical Execution Time (LET)

TDL is targeted at control applications consisting of periodic software tasks for controlling a physical environment. Thus, some tasks take information from the environment via sensors and some tasks act on the environment via actuators. Tasks that must be executed concurrently are grouped in modes. In TDL, a mode is a set of periodically executed activities that can be task invocations, actuator updates, and mode switches. A mode activity has a specified execution rate and may be carried out conditionally. TDL provides a top-level structuring unit called a module, which consists of sensors, actuators and modes that typically form a unit that delivers a specific functionality. A TDL module might be a complex combustion engine controller or a PID controller in a process automation system.

Figure 2 shows a schematic representation of a sample TDL module. The module contains one sensor variable *s1*, one actuator variable *a1*, and two modes called *main* and *freeze*. The mode *main* specifies a task invocation activity, an actuator update and a conditional mode switch, each of which must be executed once per mode period, which is every 5 milliseconds in this example. In other words, the task's LET is 5 ms. The actuator is updated with the task output value at the end of the LET. The mode *freeze* contains no activity at all.

```
module Sender {
 sensor int s1 uses getS1;
 actuator int a1 uses setA1;
 public task t1 {
  input int i;
  output int o := 10;
  uses t1Impl(i, o);
 }
 start mode main [period=5ms] {
  task [freq=1] t1(s1); //LET=5ms
  actuator [freq=1] a1 := t1.o;
  mode [freq=1] if exitMain(s1)
                then freeze;
 }
 mode freeze [period=1000ms] {}
}
```

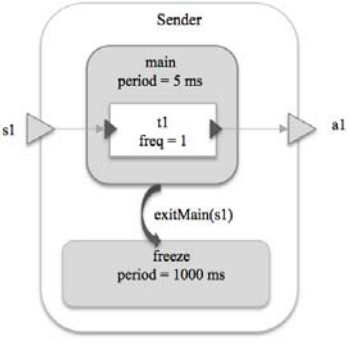


Fig. 2. TDL sample module

Executing TDL modules means executing all actions defined in the TDL code in the correct order at the specified points in time. A simulation environment for TDL modules must enforce the execution of these actions at the correct times and in this order. For a simulation time instant *t* those TDL actions are:

1. Update output ports of task invocations logically terminating at *t*.
2. Update actuators that are defined to be updated at *t*.
3. Test for mode switches that are defined at *t*. Switch mode if a switch is enabled.
4. Update input ports of tasks that are defined to be released at *t*.
5. Execute tasks that are defined to be released at *t*.

A general concept in simulating dynamic systems is that a simulation time is used to determine the actions that must be processed. TDL actions that are scheduled for a particular simulation time instant are executed without changing the time. If no actions remain, the simulation time is increased.

Consider the execution of task *t1* in the previously described example. At simulation time *0* actions in the start mode are processed. Output ports are initialized and connected actuators are updated. Sensor *s1* is read and the value is provided as input for the task, which is then executed. There are no more actions to be done at time *0*. Then the simulation time is increased to *5*, which is the end of the LET of *t1*. At simulation time *5*, output ports and actuators are updated. Next, the mode switch condition in the guard function *exitMain* is evaluated. If it evaluates to true, a mode switch to the empty mode *freeze* is performed and no further actions are processed. Otherwise the module stays in the mode and the task is executed again. The following sections describe how this general approach is implemented in Simulink and Ptolemy.

2 TDL Modeling in Simulink

The MATLAB extension Simulink from The MathWorks [7] was initially targeted for simulating control systems. It has significantly grown in popularity and due to numerous specific libraries is used for modeling systems ranging from control systems to artificial neural networks. It provides a visual, interactive environment for modeling block diagrams based on the data-flow paradigm. Simulink's model of computation is based on continuous time. This MoC is rather complex and there exists no formal definition; the implementation is hidden in the simulation engine [9, 10]. A straightforward modeling of TDL components with standard Simulink blocks is not feasible especially if they comprise several modes [6]. Simulink provides an extension mechanism by the so-called S-Function interface [8]. The subsequent section describes the Simulink integration of TDL by means of a customized S-Function and the corresponding model transformation.

2.1 LET Semantics in Simulink

The integration of TDL in Simulink requires both the modeling of TDL components, and their simulation adhering to TDL semantics. We implemented a custom Simulink block that represents a TDL module. As outlined in [6], modeling the TDL timing and data-flow relations especially for multiple modes is not feasible using standard Simulink facilities. Timing information is spread all over the model and the multitude of required signal connections makes it all but impossible to reason about or to maintain the model. It is unsolved how to obtain the exact TDL semantics in the general case of a multi-mode multi-rate system. Instead, we use a special purpose editor for describing timing aspects of the tasks, the data-flow between task ports, sensors and actuators, as well as the grouping of tasks to modes, and the mode switching logic. Tasks and guards are implemented in standard Simulink subsystems, which are referenced by the editor. In order to simulate a model, we automatically generate a simulation model consisting of standard Simulink blocks, and customized S-Function blocks.

Figure 3 shows the Simulink model for the sample TDL module from section 2. The model in Fig. 3a contains a TDL module block, a block representing the plant and a scope block to display actuator values. The content of the TDL module block in Fig. 3b shows subsystems for the task and guard function implementations and blocks for the sensor and the actuator. Fig. 3c depicts a sample implementation for the task *t1*.

To ensure that all TDL actions are executed at the correct time instants we implemented the concepts of E-Code and E-Machine in Simulink [11]. Henzinger et al. [4] introduced the E-Code concept in the realm of the Giotto project as a way of encapsulating the timing behavior and the reactivity of real-time applications. E-Code is a sequence of instructions for one period of every mode that describes the timing of all TDL actions. At run-time, these instructions are interpreted by a virtual machine, the E-Machine, which hands tasks to a scheduler or executes drivers. A driver performs communication activities, such as reading sensor values, providing input values for tasks at their release time, copy output values at their termination time, or updating actuators.

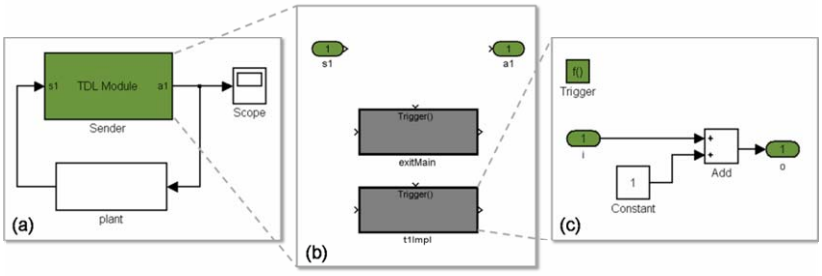


Fig. 3. (a) Simulink model with the TDL module block *Sender*, (b) contents of the Sender module block, (c) sample implementation of the t1Impl task function

For simulating the model, the TDL compiler [5] generates E-Code for all modules in the Simulink model. We implemented the E-Machine concept by using S-Functions. Drivers are generated and connected automatically. They are modeled as *Triggered Subsystems* where input ports are directly connected to output ports. This corresponds with assignments in an imperative programming paradigm as soon as the system is triggered. E-Machine blocks trigger the execution of these subsystems such that the TDL semantics are followed.

Figure 4 shows the generated simulation model for the TDL sample module *Sender*. It links with (a) the task- and guard functionality and (b) the plant model using *Goto* and *From* blocks. Section (c) contains the drivers (e.g. for reading sensor values or writing to actuators). As Simulink requires *static single assignment (SSA) form*, section (d) typically merges signals from drivers of different modes that write to the same port. As the sample module only executes activities in one single mode, signals are simply forwarded in this example. The remaining parts (e) and (f) together implement the 2-step E-Machine architecture described below.

2.2 Execution Mechanism

In typical application scenarios, TDL modules are simulated together with non-TDL controllers – typically modeled as atomic (nonvirtual) subsystems – or together with plants that don't introduce a delay. The original concept of the E-Machine [11] was adopted in order to avoid algebraic loops respectively data-dependency violations caused by Simulink's block execution strategy [12]. To solve the simulation model, Simulink derives a sorted block order based on data dependencies in the initialization phase. In this order, the simulation engine executes each block only once at a particular step. This approach is more efficient than using fixed point iteration (e.g. Ptolemy), but poses problems when trying to simulate closed control loops if no valid block order can be determined. In order to solve this and to allow Simulink to execute the plant or other blocks after actuators are updated and before sensors are read, we split duties of the E-Machine among two collaborating S-Functions (E-Machine 1 and E-Machine 2).

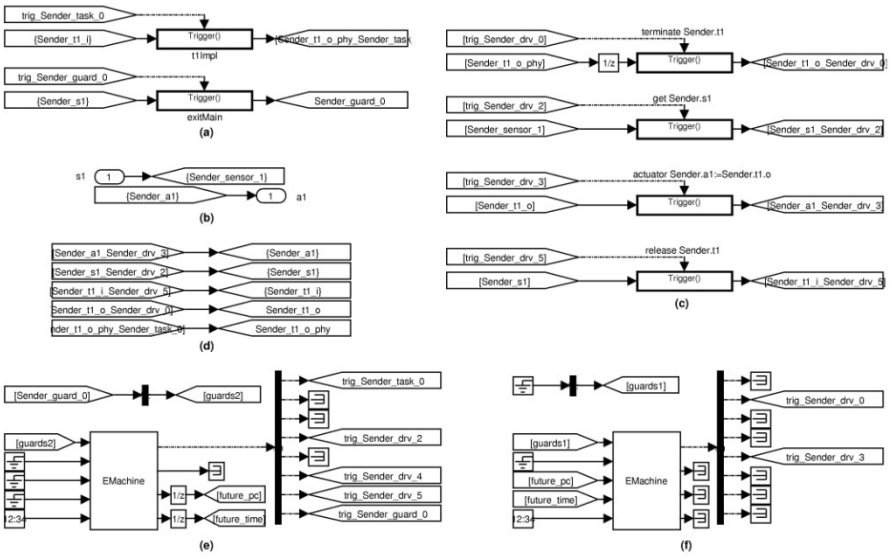


Fig. 4. The generated simulation model for the TDL module Sender in Simulink

The logical execution time of a task is controlled by the cooperating E-Machine pair. Additionally, a unit-delay block (1/z) is required for each task to indicate the fact that time passes between the task's release and terminate time. While the E-Machine ensures that the delay has effectively no impact on the timing behavior, Simulink needs it to derive a valid block update order and thus to be able to simulate the whole system.

Figure 4 show E-Machine 2 (e) and E-Machine 1 (f) with all triggers required for module *Sender*. Both E-Machines are executed at the same simulation time instants, but at different positions in the global block update order. At each simulation time, E-Machine 1 executes first to terminate tasks and update actuators, then the simulation engine executes the plant or other non-TDL blocks, and finally E-Machine 2 reads sensor values, decides on mode switches and releases tasks for the next execution. Each E-Machine executes only a subset of all drivers. Basically, E-Machine 1 executes sensor independent drivers such as task termination and actuator drivers, while E-Machine 2 executes task functions and potentially sensor dependent drivers such as mode switch or release drivers. In fact, the two E-Machines process different sections in the E-Code. We use *nop* (no operation) instructions together with an argument in order to separate the individual sections and to identify them during simulation.

E-Machine 2 signals mode switches to E-Machine 1 in order to resume with the right E-Code instructions at the next simulation step. A detailed description of the E-Machine architecture and its implementation is given in [13].

3 TDL Modeling in Ptolemy

Ptolemy II is the software infrastructure of the Ptolemy project at the University of California at Berkeley [1]. The project studies modeling, simulation, and design of

concurrent, real-time, embedded systems. Ptolemy II is an open source tool written in Java, which allows modeling and simulation of systems adhering to various models of computation (MoC). Conceptually, a MoC represents a set of rules, which govern the execution and interaction of model components. The implementation of a MoC is called a *domain* in Ptolemy. Some examples of existing domains are: Discrete Event (DE), Continuous Time (CT), Finite State Machines (FSM), and Synchronous Data Flow (SDF).

Ptolemy is extensible in that it allows the implementation of new MoCs. Most MoCs in Ptolemy support actor-oriented modeling and design, where models are built from actors that can be executed and which can communicate with other actors through ports. A Ptolemy model explaining the main Ptolemy entities is shown in Figure 5. The nature of communication between actors is defined by the enclosing domain, which is itself represented by a special actor, called the domain director. Simulating a model means executing actors as defined by the top-level model director.

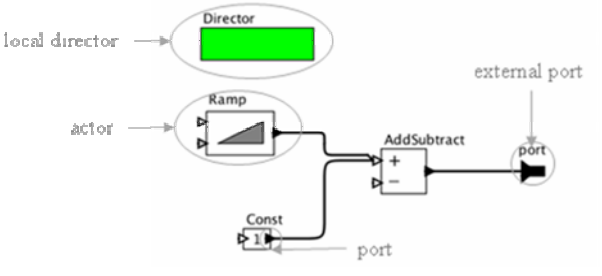


Fig. 5. Ptolemy II model

We implemented TDL as an experimental domain in Ptolemy. The implementation is based on the modal model variant of the Finite State Machine (FSM) domain in Ptolemy. Like modal models, TDL modules consist of modes with different behaviors, where only one mode can be active at a time. Transitions between states in modal models have the same behavior as mode switches in TDL.

The TDL domain consists of three specialized actors: TDLModule, TDLMode and TDLTask. The TDLModule actor (with the associated TDLModuleDirector) restricts the basic modal model behavior according to the TDL semantics. In modal models, mode switches are made whenever a mode switch guard evaluates to true whereas in TDL modules, mode switches are only allowed at predefined points in time. Similar restrictions apply to the port updates. To ensure LET semantics of the tasks, input ports of TDL tasks are only allowed to be read once at the beginning of the LET, output ports are only allowed to be written at the end of the LET and not when a task finished its computation. TDL requires a deterministic choice of simultaneously enabled transitions, which is not provided by the FSM domain. In this respect, we define an order on all transitions from a mode and take the first enabled transition in this order. TDL timing information such as the mode period is associated with TDL actors in the model.

TDL activities are conceptually regarded as discrete events that are processed in increasing time stamp order. Thus, a TDL module can be seen as a restricted DE actor. This enables the usage of TDL modules inside every domain that can deal with DE actors. The example from section 2 modeled in Ptolemy is shown in Figure 6.

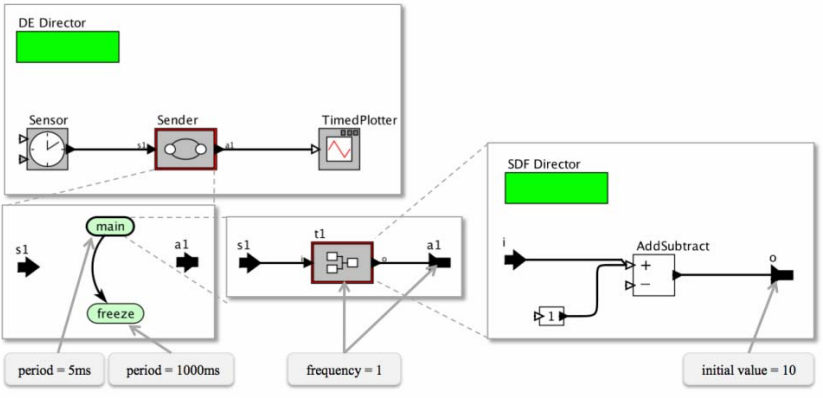


Fig. 6. A TDL module in Ptolemy

The model shown in the top left box of Figure 6 contains a TDL Module and two actors to provide sensor values and display actuator values. The TDL Module contains two modes (see Figure 6, the lower left box). Both modes have the period associated as a parameter. The main mode contains the task and the association of sensor and actuator values to input and output ports of the task. The frequency of task invocation, which determines the LET, is defined as a parameter. The task is an SDF actor, which executes in logically zero time. The top-level director is a DE director. The DE director uses a global event queue to schedule the execution of actors in the model. The TDL module places events in this queue for every time stamp where at least one TDL action is scheduled.

4 Summary and Conclusions

We presented the modeling and simulation of TDL real-time components with logical execution time (LET) in two different simulation environments, namely Simulink and Ptolemy. Due to fundamental differences in the simulation environments we had to apply two different approaches regarding both, modeling and simulation. Table 1 lists the main concepts in modeling and simulation of TDL modules and how they were implemented in Simulink and Ptolemy.

The open and extensible architecture of Ptolemy allowed us to express the complete TDL semantics in the model. In the Simulink integration, the developer only models the functionality (task implementations and the plant) with Simulink blocks. Timing, mode switching logic and the overall application data-flow are described with a special purpose editor. An elaborate model transformation automatically creates a simulation model that contains data-flow and timing information.

Both approaches extend the existing simulation framework with a new actor (block) containing tasks with real time requirements described by TDL. In Simulink, the timing requirements are enforced by the E-Machine, in Ptolemy the director of the TDL domain employs the TDL semantics.

Table 1. Comparison of TDL Modeling and Simulation in Simulink and Ptolemy

	Simulink	**Ptolemy**
Modeling		
TDL Module	A TDL module block (actor) was implemented and is available in a library.	
TDL Mode	Modes are defined in the TDL editor.	A TDL mode actor was implemented and is available in a library. A TDL mode contains input and output ports of a TDL module (=sensors, actuators) and TDL tasks.
Data-flow (Connections between sensors, task ports, and actuators)	Data-flow is defined in the TDL editor.	Graphically connect TDL module ports to task ports.
Task functionality	An embedded model (subsystem) implements the task functionality.	
	Stub subsystems are generated, which provide the input and output ports. They have to be implemented with Simulink blocks.	A TDL task actor was implemented and is available in a library. The TDL task is a composite actor containing the embedded model.
Simulation		
Triggering of all actors (blocks) in the model (TDL modules and plant)	Simulink triggers all blocks of the plant and the E-Machine S-Function blocks.	The top-level director, which has to be able to deal with DE actors, triggers all actors including TDL modules.
Timing description of TDL actions	Generate static E-Code before starting the simulation.	Generate events dynamically during the simulation.
Enforcing the timely execution of TDL actions	The E-Machine interprets E-Code instructions and triggers TDL actions.	The TDL Director creates events for all TDL actions.

Ptolemy uses events to schedule TDL activities dynamically whereas in Simulink, we compute a static list of TDL activities in the form of E-Code before starting the simulation. The main advantage of the static approach is its low computational overhead for determining the next TDL actions. Maintaining an E-Code program counter is enough, it is not necessary to create events and handle dynamically changing event queues. Event queues, on the other hand, potentially require less storage space than E-Code, because they contain only the immediate follow-up events, not all activities of a mode period.

References

[1] Brooks, C., Lee, E.A., Liu, X., Neuendorffer, S., Zhao, Y., Zheng, H. (eds.): Heterogeneous Concurrent Modeling and Design in Java, Introduction to Ptolemy II. vol. 1. EECS Department. University of California, Berkeley, UCB/EECS-2007-7 (January 2007)
[2] Templ, J.: TDL - Timing Definition Language 1.4 Specification (2007), http://www.preetec.com/
[3] Henzinger, T.A., Kirsch, C.M., Sanvido, M., Pree, W.: From Control Models to Real-Time Code Using Giotto. IEEE Control Systems Magazine 23(1) (2003)
[4] Henzinger, T.A., Kirsch, C.M.: The Embedded Machine: Predictable, portable real-time code. In: Proc. of the PLDI. ACM Press, New York (2002)
[5] preeTEC, http://preetec.com/
[6] Pree, W., Stieglbauer, G.: Visual and Interactive Development of Hard Real Time Code. In: Automotive Software Workshop San Diego (ASWSD) (January 2004)
[7] The MathWorks, http://www.mathworks.com
[8] The MathWorks. Simulink 7, Writing S-Functions (2008)
[9] Carloni, L., Benedetto, M.D.D., Passerone, R., Pinto, A., Sangiovanni-Vincentelli, A.: Modeling Techniques, Programming Languages and Design Toolsets for Hybrid Systems, Project IST-2001-38314 COLUMBUS-Design of Embedded Controllers for Safety Critical Systems, WPHS: Hybrid System Modeling, version: 0.2, Deliverable number: DHS4-5-6 (July 2004)
[10] Baleani, M., Ferrari, A., Mangeruca, L., Sangiovanni-Vincentelli, A.L., Freund, U., Schlenker, E., Wolff, H.-J.: Correct-by-Construction Transformations across Design Environments for Model-Based Embedded Software Development. In: Proc. of the Conf. on Design, Automation and Test in Europe, DATE (2005)
[11] Stieglbauer, G.: Model-based Development of Embedded Control Software with TDL and Simulink. PhD thesis, Department of Computer Sciences. University of Salzburg, Austria (2007)
[12] Mosterman, P.J., Ciolfi, J.E.: Interleaved execution to resolve cyclic depen-dencies in time-based block diagrams. In: Proc. of the 43rd IEEE Conference on Decision and Control, CDC (2004)
[13] Naderlinger, A., Templ, J., Pree, W.: Simulating Real-Time Software Components based on Logical Execution Time. In: SCSC 2009: Proceedings of the 2009 Summer Computer Simulation Conference (2009)

Requirements for Service Composition in Ultra-Large Scale Software-Intensive Systems

Claudiu Farcas, Emilia Farcas, and Ingolf Krüger

University of California, San Diego, USA
{cfarcas,efarcas,ikrueger}@ucsd.edu

Abstract. Ultra-Large Scale Software-Intensive Systems (ULSSIS) are integrated networks of capabilities that serve large communities of stakeholders and have a broad spectrum of crosscutting concerns. The added value of an ULSSIS emerges from the interplay of a set of constituent systems, both existing and emerging systems. We see the systems-of-systems integration challenge as a major concern for ULSSIS. System integration has to address systematic and seamless composition of services, consider upfront crosscutting concerns such as policy and security, assure system dependability and quality of service, and show agility to deal with changes in the requirements and the environment in which the ULSSIS operates. In this paper, we present some of the challenges for service composition, propose an architectural style for addressing them, and discuss the experience gain in designing an ULSSIS, namely the infrastructure for ocean observatories.

1 Introduction

The technological advances of the past decades point us to an exciting challenge, namely, ultra-large scale software-intensive systems (ULSSIS). These systems are large in multiple dimensions, such as diverse stakeholder communities, number of distributed components that have to be integrated, interdependencies between components, quality of service, storage and computational infrastructure, to name just a few. ULSSIS scalability requirements go beyond what current software and system engineering techniques can address, and require novel approaches for analysis, design, deployment, and evolution. As ULSSISs are often built from existing systems as well as emerging systems, the requirements for them cannot be comprehensively specified upfront; moreover, requirements for the constituent subsystems evolve independently from the composite. Therefore, no methodology will support ULSSIS to be built right and complete in a single exercise (there is no silver bullet [1]); instead, these systems should be designed such that they adapt over time to fit the changing environment and meet the expected quality of service.

Moreover, the complexity of systems of systems results in challenges that go far beyond the purely technical ones – ULSSISs are socio-technical ecosystems [2] involving software-intensive systems, organizations, policies, and economics. Thus, ULSSIS comprises a set of dynamic, interdependent communities that collaborate or compete for resources. [2] describes ULSSIS as having the following characteristics:

C. Choppy and O. Sokolsky (Eds.): Monterey Workshop 2008, LNCS 6028, pp. 93–115, 2010.

(1) decentralization regarding data, development, evolution, and operational control; (2) inherently conflicting, unknowable, and diverse requirements due to the variety of stakeholders; (3) continuous evolution and deployment, as new capabilities need to be integrated into an ULSSIS while it is operating; (4) heterogeneous, inconsistent, and changing elements – parts have different providers, and will evolve asynchronously; (5) erosion of people/system boundary – people will not be just users, but will affect the emergent behavior of the ULSSIS; (6) failures will be the norm – they must be anticipated and tolerated; (7) new paradigms for acquisition and policy.

The added value of an ULSSIS emerges from the interplay of a set of constituent systems; the interplay often combines capabilities of the constituent systems in a novel, typically value-added, often surprising way. ULSSIS constituents are usually designed and developed by different organizations with little or no awareness that the constituents' capabilities would later be composed as parts of a larger ULSSIS system. As each organization has its own boundaries and policies, the access to resources in these organizations must be governed. For instance, for the integrated ULSSIS, policy monitoring and enactment has to be enabled across the overall architecture rather than by providing ad-hoc solutions for all end-points. We see the systems-of-systems integration challenge as a major concern for ULSSIS. System integration has to address the composition of systems into an integrated network of capabilities, consider upfront negotiations and crosscutting concerns between systems, assure system dependability and quality of service, and foster agility to deal with changes in the requirements and the environment in which the integrated system operates. The more flexible the architecture is with respect to updating or substituting existing subsystems, the easier the ULSSIS will adapt to changes, which is particularly important in the context of dynamic system reconfiguration.

Given the ULSSIS size and massive distribution, it is practically impossible to describe all the behaviors of all components involved completely. Often, we only have a partial view on the requirements of the overall system. Services as partial system behaviors and corresponding Service-Oriented Architectures (SOAs) have emerged as an accepted solution to assembling large distributed heterogeneous systems out of diverse services. SOAs can use standards-based infrastructure to create ULSSIS out of loosely coupled, interoperable services by mapping existing systems into services, then orchestrating communication between the services. Nevertheless, as proven by current state-of-the-art, just using an SOA is not sufficient to solve the multitude of problems in ULSSIS, such as governance and constantly changing requirements, implied by multiple stakeholders with many business concerns. Each stakeholder brings its own business processes, capabilities, and requirements to the integration task. The integration of these concerns requires a scalable framework that provides decoupling between the various concerns and allows for system-of-systems integration and hierarchical decomposition according to one or more concerns independent of the other concerns.

In this article, we shed light on the requirements for service composition in ULSSIS using the recently developed Rich Services framework [4] as a promising solution and the Ocean Observatories Initiative (OOI) – CyberInfrastructure (CI) [3] of the National Science Foundation (NSF) as an example of its application. OOI combines oceanographic instrument and sensor and actuator networks, data and computation grids, and a broad set of end-user applications in a CyberInfrastructure (OOI-CI)

with novel capabilities for data distribution, modeling, planning and control of oceanographic experiments. OOI has a vast set of stakeholders, ranging from sponsors to ocean scientists to architects and implementers to operators and maintainers to the general public. The OOI-CI resources are distributed both physically and virtually among different organizations, each with their own policies for resource access and data delivery or consumption. OOI is an excellent example of ULSSIS; it places tremendous demands on the underlying integration fabric in terms of governance, security, dependability, flexibility, maintainability, and other quality properties.

Outline. The remainder of this paper is structured as follows. In Section 2, we present the requirements for service composition in ULSSIS as informed from our experience in the OOI project. In Section 3, we motivate the need for a service-oriented approach in designing and building ULSSIS. In Section 4, we present the main entities of the Rich Services framework and show how it can be used to address the challenges identified in Section 2. In Section 5, we present some aspects of the OOI CI and how we applied Rich Services to this case study. Then, we present lessons learnt from our experience in OOI CI. A discussion of related work and conclusions round out the paper.

2 Challenges for Service Composition in ULSSIS

The value and complexity of an ULSSIS emerges from the integration of individual subsystems into massively distributed large-scale systems of systems, networked federations of capabilities that are exposed to various communities of interest. Key challenges of composing the services forming the ULSSIS include understanding and modeling both the existing constituent systems and the stakeholder requirements; designing an extensible architecture for such a massive integration that meets the stakeholder's needs; yet, creating systems that are robust, performant, and maintainable; and evolving the overall system as requirements themselves evolve. In the following, we identify central requirements regarding service composition for ULSSIS and detail the challenges they present from a software engineering perspective.

Evolution and adaptability at all scales represents one of the fundamental characteristics of an ULSSIS. Given their size and complexity, ULSSIS such as OOI-CI, do not follow a single regular design/deploy/operate pattern as a whole; instead, they evolve continuously through successive iterations along multiple dimensions. In this sense, we could consider them as giant cyber-organisms that continuously grow new capabilities of various kinds in relationship with the environment in which they operate. Consequently, classic waterfall development processes are hardly applicable; instead, agile development processes [5] with several iterations and corresponding system "releases" are required for dealing with emergent, evolving, and ever-changing requirements that affect the ULSSIS in parts or as a whole.

The management of distributed resources is one of the key requirements for designing and building ULSSIS. Taking as example the OOI-CI, we have to deal with geographically distributed resources spanning from the entire continental US to further global instrument platforms in extreme remote locations within the Pacific Ocean. In addition, we have to take into account the possibility of disconnected and autonomous operation of these resources based on the availability of radio/satellite

links and field-determined operation plans. Hence, dynamic addition of new capabilities and resources is critical for the operation of an ULSSIS; approaches using static system models with predefined components can hardly face this challenge.

Federated decentralized operation is a key requirement implied by the number of business concerns of the vast number of stakeholders in ULSSIS. Given the various factors that differentiate an ULSSIS, from a typical computation infrastructure of today, such as massive distribution, platform diversity, overall cost, it is reasonable to assume that most ULSSIS have policy, governance, and regulatory constraints on each of their constituents. Identity, ownership, authority domains, and operational domains are just a few of the concepts of the OOI-CI example that must have equivalent models in the overall system architecture. OOI resources, for instance, belong to the organization (read: university and its principle investigator) who has created and deployed them; on the other hand, by becoming part of the OOI infrastructure these resources must now become accessible to participants of the infrastructure. Policies govern the circumstances under which the resources join and leave the infrastructure, how they can be discovered, accessed, and utilized.

High availability with active failure management is a critical requirement for the on-going operation of an ULSSIS. As we expect future ULSSIS to continuously evolve, we also have to expect that failures will be the norm [2], and not exception in its operation. Hence, all its constituents must be designed to operate under the assumption that other parts of the system can fail, or become unavailable. Scalability in this sense implies redundant safety-critical capabilities and fail-over mechanisms to maintain the system under operation. In the case of OOI-CI, the disconnected operations of some resources may require mechanisms to mask their unavailability to the rest of the system.

Scalable computational and data storage infrastructure. In the particular case of the OOI-CI, the ability to scale data gathering/fusion/distribution and advanced ocean-modeling experiments well past current computational limitations is of paramount importance. This requirement translates into abstracting from the complexity of the hardware and networking infrastructure and exposing their services in a transparent way. In a very real way ULSSIS share many characteristics with operating systems; this leads to an understanding of "cyberinfrastructures" as "wide-area operating systems." We note that most ULSSIS are built on top of some notion of cyberinfrastructure.

The management of distributed state and shared understanding of the semantics of the message pattern implementing the intended behavior is also of significant importance to the correct operation of an ULSSIS. Specifically, the integration challenge is largely solved via the interplay of the constituent subsystems. When the number of stakeholders is significant such as in the case of the OOI-CI, this interplay (the "business logic") often requires negotiation phases, shared understandings, confirmed intent, and clear commitments that must be supported by the ULSSIS architecture. This is especially true when dealing with concepts such as policies and governance that may span well over multiple parts of the system driving the overall system behavior and its interaction with the environment including end-users.

Extensive and expressive communication with the environment is the base of the usability of a ULSSIS. For instance, in the case of the OOI-CI, the environment

will be sampled through a wide range of sensor networks and domain specific instruments. Computation results and mission plans for gliders would provide the feedback of such ULSSIS into the environment. In addition, an important constituent of an ULSSIS are its users, which according to specific criteria, would form virtual communities of interest (VCoI) around the offered capabilities. VCoI members share knowledge as intellectual property (along with the associated data), drive the day-to-day operation of the system, request observation plans, and keep the system running.

Far from extensive, this set of requirements briefly touches upon the challenges of designing an architecture capable of supporting ULSSIS. The continuous evolution of such systems implies a continuous evolvement of their architecture in multiple iterations. We believe that SOAs may be up to the challenge, especially when used within frameworks such as the Rich Services (detailed in Section 4) and associated agile development processes. Nevertheless, establishing and managing in the long run the development practices by which these large-scale integrated systems evolve is a fundamentally new challenge for Software Engineering.

3 Why Services?

It is fair to say that Service-Oriented Architecture (SOA) and Service-Oriented Development have become widely and successfully used approaches to addressing the inherent complexities of ULSSIS. We follow [6] in defining SOA as "A paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations." This directly speaks to many of the requirements we have identified in the previous section. In this definition, services are manifestations of capabilities – either in the form of "constituent" capabilities that are "organized and utilized", or the composite capability that emerges from the organization/utilization.

The ability to "discover" services at both design- and runtime, together with clearly specified syntactic service interfaces (similar to APIs, "Application Programming Interfaces" formerly known from component-oriented development) leads to the benefits of loose coupling, encapsulation, flexible composition, scalability, and reuse, commonly associated with SOAs. Specifically, Web Services [7] as the prime implementation choice for SOAs offer a set of open, core technology standards that enable developers to reap these benefits. The core technologies are solely concerned with the (1) the specification of service interfaces, (2) offering and discovery of services, and (3) the binding to and communication with services.

Of course, similar benefits and technology solutions were promised already by structured analysis and design [8], object-oriented [9] and later component-oriented development approaches and infrastructures [10]. However, SOA and its associated technologies have proven successful over their predecessors.

Structured analysis and design favors a top-down, hierarchical decomposition of complex systems with little concern or opportunity for dynamic discovery or sharing of capabilities across subsystems. Furthermore, the strict hierarchical decomposition results in significant redesign effort if capabilities change as requirements change. This is especially true if cross-cutting concerns, such as authentication, authorization,

logging, failure management, policy and governance (to name but a few), which are often discovered late in the development process, yet, may have profound impact on the resulting system architecture.

SOA, in contrast, emerges from the composition of loosely coupled services; this composition can be wired at design-time, or at runtime, such that flexible configuration and reconfiguration can occur as requirements change. This facilitates a combination of top-down and bottom-up design.

Object- and component-oriented development approaches, with CORBA as the flagship implementation and deployment infrastructure, did enable flexibility in composing capabilities across heterogeneous, distributed computation nodes. Furthermore, there were explicit mechanisms for infrastructure services addressing the cross-cutting concerns mentioned above. However, these approaches were challenged by inadequately chosen levels of granularity (capability bloat) per component (typical components would implement many services) hindering finely granular reuse, as well as by the use of proprietary protocols for discovery, access and information exchange, hindering interoperation across technology platforms and suppliers. The latter, of course, largely defeated the purpose of having an integration platform in the first place.

SOA and its core Web Service implementation technology standards, on the other hand, build on open web-based standards for service interface specification, service publishing, discovery, access, and information exchange. The infrastructure focus on these (rather than on a plethora of infrastructure services as was the case in CORBA's days) has lead to genuine wire-level interoperability.

However, this advantage came at the expense of "exploding" the cross-cutting concerns into separate, still evolving technology standards [7] that have yet to converge/stabilize. In particular, there is no integrative "service composition" approach yet that combines the core technical interoperability with the management of cross-cutting concerns. Furthermore, we are only beginning to develop an understanding of how to slice large-scale systems into SOAs, and what the successful architectural design and implementation patterns for these architectures will look like.

To facilitate this process, we have developed an architectural blueprint, called "Rich Services" that combines the flexibility of SOAs with the scoping and systematic development processes of its predecessors.

4 Rich Services

The number and complexity of various business, functional, and non-functional concerns that need to be addressed for an ULSSIS create a strong demand for a richer service-oriented framework that is scalable, dynamic, and provides decoupling between various concerns. We distinguish between "horizontal" and "vertical" service composition. Horizontal service composition refers to managing the interplay of services and the corresponding crosscutting concerns at the same logical or deployment level. On the other hand, vertical service composition refers to the hierarchical decomposition of one service (and the crosscutting concerns pertaining to this service) into a set of sub-services such that their environment is shielded from the structural and behavioral complexity of the embedded sub-services and the form of their composition.

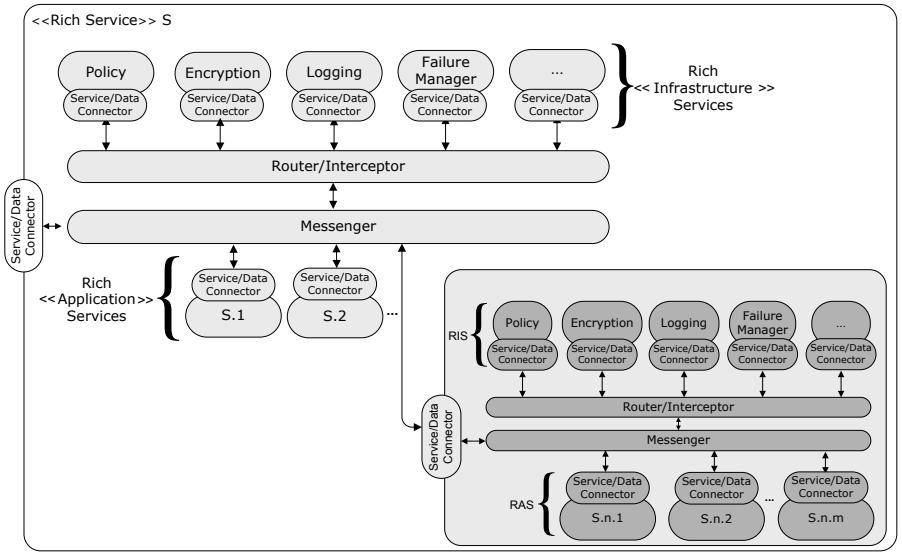


Fig. 1. Rich Services architectural pattern for service composition

For this purpose, we developed the Rich Services architecture [4] as a type of SOA that provides decoupling between the constituents of an ULSSIS and allows for hierarchic service composition according to one or more concerns independent of the others concerns. Taking each concern in isolation, the Rich Services framework allows for creating various projections of the overall ULSSIS model particularly tailored for dealing with that concern.

The main entity of the architecture is the notion of Rich Service. A Rich Service could be a simple functionality block (such as a Web service), or it could be hierarchically decomposed. A Rich Service is composed of several entities: (a) the Service/Data Connector [4], which serves as the sole mechanism for interaction between the Rich Service and its environment, (b) the Messenger and the Router/Interceptor, which together form the communication infrastructure, and (c) the constituent Rich Services connected to Messenger and Router/Interceptor, which encapsulate various application and infrastructure functionalities.

To address the horizontal integration challenge, the logical architecture is organized around a message-based communication infrastructure that enables loose coupling between the services implementing the system's business logic. The Messenger layer is responsible for message transmission between endpoints. By providing the means for asynchronous messaging, the Messenger supports decoupling of Rich Services. The second layer, the Router/Interceptor, is in charge of intercepting messages placed on the Messenger, then routing them. The routing policies of the communication infrastructure are the heart of the Router/Interceptor layer. Leveraging the Interceptor pattern [41] readily facilitates dynamic behavior injection based on the interactions among Rich Services. This is useful for the injection of policies governing the integration of a set of horizontally decomposed services.

To address the vertical integration challenge, we distinguish between *Rich Application Services* and *Rich Infrastructure Services* [4], where both kinds can be further hierarchically decomposed. Rich Application Services interface directly with the Messenger; they have no influence on how messages are routed. In contrast, the Rich Infrastructure Services interface directly with the Router/Interceptor and can change the routing of messages. The Service/Data Connector encapsulates and hides the internal structure of the connected Rich Service, and exports only the description and interfaces that the connected Rich Service intends to provide and make visible externally. The communication infrastructure is only aware of the Service/Data Connector, and does not need to know any other information about the internal structure of the Rich Service.

Rich Infrastructure services may be used to orchestrate the business flows between the core functionality exposed as Rich Application services, through specialized interceptors that handle specific concerns. In this way, there is a clear separation between the business logic and its external constraints, whereas the composition between them happens at the infrastructure level. For instance, support for system-wide encryption/decryption of the messages can be easily added by using a RIS that (1) intercepts messages and routes them through an encryption service, then (2) sends them to the destination service, where (3) it intercepts the encrypted messages and routes them through a decryption service, before handling them to their final destination. This approach avoids implementing such functionality in each RAS and improves system flexibility (e.g., the encryption algorithm can be easily changed without influencing existing RAS functionality).

We use Rich Services architectural blueprint both as a logical model and as a guide to a deployment architecture leading to an Enterprise Service Bus (ESB) solution. ESB technologies such as the Mule ESB framework [15] provide message routing amongst well-defined, loosely coupled, coarsely granular services similar to our notion of Rich Services. The close alignment of the logical architecture for the Rich Services and the ESB deployment architecture yields a direct logical-to-deployment concept mapping, though other deployment architectures would also work.

In the following paragraphs, we show how Rich Services address the ULSSIS service composition challenges identified in Section 2.

Evolution and adaptability at all scales. SOAs provide loose coupling between system components and support both agile development and independent development of the parts of the system. In particular, the iterative Rich Services development process [13] enables changing requirements at all stages of the development, which can be integrated into the overall system architecture in future iterations. The development process promotes encapsulation, separation of concerns, reusability, and service-orientation, where emerging requirements can often be implemented with minimal or no changes to working code. Service recomposition to accommodate changed business logic can happen either at the infrastructure or the application service level. If it happens at the infrastructure level, the core business application code need not be affected by the change; this allows modifications transparent to the services that are being composed. The Rich Services infrastructure enables dynamic changes to business flows by intercepting and transforming message flows, thereby exploiting the loosely-coupled services and leveraging the routing capabilities of SOA platforms such as ESBs.

Management of distributed resources. SOAs address these concerns by exposing the capabilities of resources through services that can be discovered and used dynamically during system operation. In particular, the Rich Services framework goes further in supporting dynamic service composition by promoting hierarchical decomposition and encapsulation of local business logic at all composition levels. For disconnected operations, the Service/Data Connector of each resource may either notify other services about the unavailability of the resource, or act as a service proxy and until the resource becomes again available. Hence, upper level services may choose between resources with similar capabilities based on their availability and mask the temporal unavailability of some resources.

Federated decentralized operation. Rich Services enable us to model these concerns in isolation and integrate them as needed as infrastructure services at the corresponding level of hierarchy in the overall ULSSIS model. The Rich Services framework, through the use of the Router/Interceptor layer, reduces dependencies between services and their relative locations in the logical hierarchy. For instance, every Rich Service can be part of a different authority domain and have its own internal policies and resources. The communication between various services from different authority domains is enabled via infrastructure services in the higher level of the hierarchy, such as a policy service, encryption service, or authentication service. Concerns such as publishing and service discovery can be represented via additional services connected to the communication infrastructure at different levels of the hierarchy.

This approach enables services from different levels of a hierarchy, possibly with different properties (such as encryption and security requirements) to interact with each other seamlessly without ever being aware of such incompatibilities. The Router/Interceptor, with the help of additional intermediary infrastructure services, is the means for providing such a seamless communication.

High availability with active failure management. A Rich Services-based implementation is particularly adapted to this mode of operation as its routing/interceptor layer can dynamically reroute messages to replicas or proxies of failed/unavailable services without interrupting the business logic flow. In this sense, inherently hierarchical by nature, the Rich Services design pattern breaks the fault chain by providing uninterrupted capabilities to any upper level service. In previous work, we introduced a comprehensive approach [11] towards application-level fault tolerance for service-oriented applications by combining the RS architectural pattern with a model-based approach to failure specification, detection and mitigation. We use a similar approach for ULSSIS by focusing on the interactions between the system components for specifying and enforcing failure management. Formal models based on interaction specifications provide the means to specify communication patterns between different subsystems. Together, those patterns capture the full behavior of the system. A policy-based mitigation allows us to support an expandable list of recovery strategies that are dynamically chosen and applied to provide high-availability.

Scalable computational and data storage infrastructure. In a Rich Services-based implementation, any Service/Data Connector abstracts the implementation, business logic, and platform details of its Rich Service. Coupled with virtual machine technologies, even the platform-specific requirements of an implementation of a “leaf”

Rich Service become irrelevant; hence, computational capabilities can easily scale with the demand. Data transformation capabilities of the Service/Data connector also enable complete abstraction of the storage mechanisms implemented by a Rich Service; thus, a simple database would be indistinguishable from a large cluster of databases. This translates into major cost savings when using COTS components [12] for building the underlying physical infrastructure of the ULSSIS according to the Rich Service architectural pattern.

The management of distributed state. Reliable messaging, replication, encapsulation of business logic, and orchestration of interactions are just a few of the elements that a Rich Service-based solution would employ to achieve this objective.

Extensive and expressive communication with the environment. The Rich Services framework helps in these scenarios by abstracting from domain-specific entities such as physical instruments, providing uniform ways for accessing capabilities, providing support for ad-hoc dynamic reconfiguration of services, grouping of services and data for community collaboration in hierarchical structures.

5 Case Study: Ocean Observatories

We demonstrate the utility of the proposed architecture blueprint by using a case study from the domain of Earth sciences, namely the ongoing NSF Ocean Observatories Initiative (OOI) program [3]. The main goal of the OOI is to provide the basic infrastructure for sustained, long-term oceanographic and climate-change research. It comprises three types of interconnected ocean observatories spanning global, regional and coastal scales. The core capabilities and the principal objectives of ocean observatories participating in the OOI program are collecting real-time data, analyzing data and modeling the ocean on multiple scales, and enabling adaptive experimentation within the ocean.

The OOI CyberInfrastructure (CI) constitutes the integrating element that links and binds the observatories and associated sensors into a coherent large-scale system-of-systems. CI also enables direct access to instrument data acquisition and control, and the opportunity to seamlessly collaborate with other scientists, institutions, projects, and disciplines.

A traditional data-centric CI, in which a central data management system ingests data and serves them to users on a query basis, is not sufficient to accomplish the range of tasks ocean scientists will engage in when the OOI is fully implemented. Instead, a highly distributed set of capabilities are required to facilitate: end-to-end data preservation and access; user-driven and automatic control of how data are collected and analyzed; direct, closed loop interaction of models with the data acquisition process; virtual collaborations created on demand to drive data-model coupling and share ocean observatory resources (e.g., instruments, networks, computing, storage and workflows); end-to-end preservation of the ocean observatory process and its outcomes; automation of the planning and prosecution of observational programs.

This case study is an elaboration of the OOI-CI Final Network Design [3]. Clearly, in this article we can only scratch the surface of the complexity of building an architecture of the scale of OOI. However, this case study allows us to show how we address the ULSSIS concerns by decomposing the OOI-CI services both horizontally

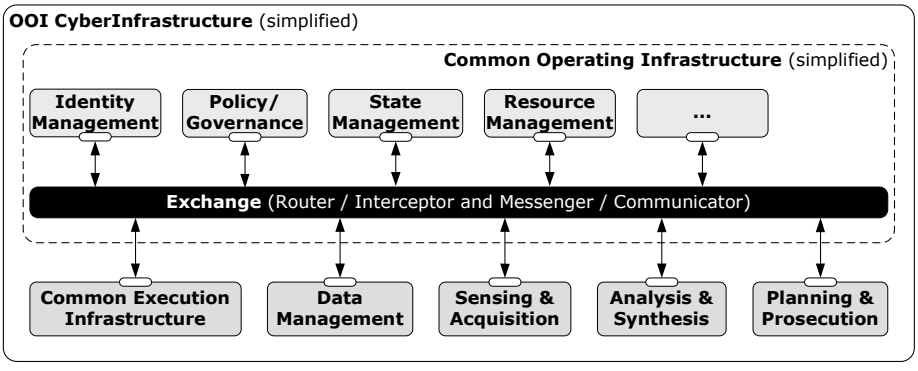


Fig. 2. OOI-CI decomposition according to business concerns

and vertically through the Rich Services architecture blueprint, and to express valuable lessons learned in the interaction with a multi-disciplinary set of stakeholders.

OOI-CI is a complex ULSSIS with functionality covering multiple business concerns. Figure 2 shows a simplified decomposition of the OOI-CI into six subsystems, i.e., collections of services pertaining to different business concerns. Four subsystems (namely, Sensing and Acquisition, Data Management, Planning and Prosecution, Analysis and Synthesis) address the oceanographic science- and education-driven operations of the OOI Integrated Observatory. The other two subsystems provide the infrastructure services: the Common Operating Infrastructure provides the distributed, message-based, service-oriented integration and communication infrastructure, whereas the Common Execution Infrastructure provides the virtualization of computational and storage resources.

Evolution and adaptability at all scales. Large-scale projects such as OOI are subject to requirements changing over time. Not only do user requirements evolve, but the technologies used in implementing the subsystems do, too. Rich Services integration strategy allows constituent subsystems to evolve independently from the composite system. This also sets the stage for dynamic system reconfiguration to adapt to changes in the environment or in the requirements. Subsystem functionality is exposed to the OOI network as services with defined access interfaces, and the only way of interacting within the OOI network is by messages. Service-orientation and messaging realize loose coupling of components, resulting in flexibility and scalability. For manageability, services are grouped according to business concerns into subsystems, which can be developed independently.

Extensive and expressive communication with the environment. The environment of the OOI ULSSIS involves both users and physical sensors. The Sensing and Acquisition subsystem provides the means for interfacing with oceanographic sensors (e.g., for temperature, pressure, salinity), which may be aggregated into research instruments deployed on instrument platforms. This subsystem also deals with providing power ports, connectivity, and other physical observatory infrastructure and provides state-of-health monitoring and oversight over the data acquisition process. The Planning and Prosecution subsystem coordinates all scientific activities. It provides

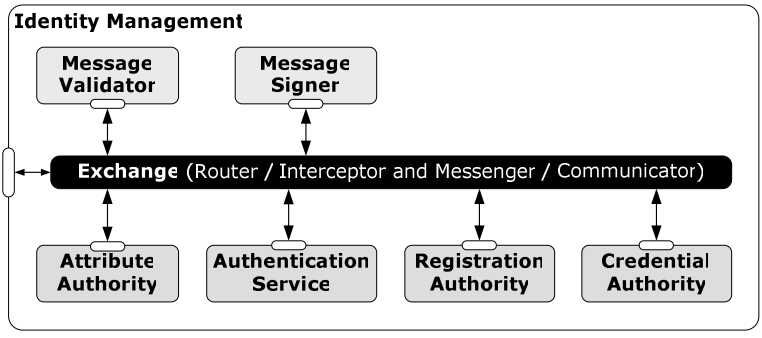


Fig. 3. OOI-CI > COI > Identity Management service decomposition

resource planning and plan execution, including the development, refinement, configuration, and enactment of observation plans. It also provides the framework for autonomous glider control.

From the user perspective, the Analysis and Synthesis subsystem provides a set of basic frameworks to define data analysis and manipulation processes based on user-provided workflows and computation. In addition, this subsystem provides the basis for virtual collaborations, interactive analysis, and visualization. The Data Management subsystem provides access to various information products together with their associated metadata. For processing and manipulation of data artifacts, it makes use of ontology-based mediation to transform information between different syntactic and semantic representations.

Scalable computational and data storage infrastructure. The Common Execution Infrastructure subsystem handles the scheduling, provisioning and execution of all necessary OOI-CI computations. From a deployment perspective, it uses virtual machines that abstract from the underlying computational resources. Dynamic provisioning of virtual machines inside a computing cloud enables scalable execution of the business logic according to the actual demand.

The Common Operating Infrastructure (COI) subsystem provides a message-based exchange (including the router/interceptor and messenger/communicator capability) between all services, and ensures pervasive and consistent governance/policy enforcement, identity management, and resource management. It also allows subsystem services to be composed to handle complex interactions, and manages the overall service orchestration. At all abstraction levels, infrastructure services plugged into the Exchange service can modify the interaction patterns by re-routing, filtering, or modifying the exchanged messages. This feature allows the validation and signing of messages and also injection of policies governing the integration of a set of services. For instance, Figure 3 depicts a decomposition of the Identity Management service of the COI subsystem. The Message Signer service signs all messages, whereas the Message Validator validates the identities of the sender and intended recipient from an incoming message through the Authentication Service and the Credential Authority, and checks the validity of the message against a set of known interaction patterns via the Authentication service and the Attribute Authority. The Authentication Service, Registration Authority, Credential Authority and Attribute Authority provide the core

services of an Identity Provider, which is a critical component for a scalable infrastructure with federated decentralized operations.

Management of distributed resources and their states. The Rich Services framework provide resource location independence; user applications are shielded from the complexity of the system and where resources are located. For OOI-CI, the COI subsystem provides a Resource Management service, which enables seamless utilization of resources across the entire CyberInfrastructure (see Figure 4). The Resource Repository service provides references to all resources known to the OOI-CI. Through the Resource Integration service, resources can participate in interaction patterns implemented by OOI services (e.g., a storage resource may be used to record states of various services), or may provide their own services (e.g., a glider may act as an instrument platform and proxy for the sensors that it carries). The Resource Collaboration service provides the collaboration framework between different facilities and the sharing of resources within the OOI federation. The Resource Lifecycle service provides the means to track and manage resources throughout their entire lifecycle from development to decommissioning.

Federated decentralized operation. A system satisfying the goals of OOI would support scientific discovery by providing eligible oceanographers ubiquitous access to instrument networks for sensing and actuation, computational resources, and modeling and simulation facilities, as well as means for distributed data storage and access. A traditional SOA approach would quickly reach its limits in the face of the challenges induced by the diverse requirements of supporting governance of the different authority domains, access policies, and concerns of the multiple stakeholders involved in such a complex system-of-systems. The complexity of the resulting cyber-infrastructure requires a decomposition methodology and an architecture that supports the deployment, operation, and distributed management of thousands of independently owned taskable resources of various types (e.g., sensors, sensor platforms, processes, numerical models and simulations) across a core infrastructure operated by independent stakeholders.

In the case of the OOI-CI, the Router/Interceptor layers and the Service/Data connectors of each Rich Service enable the composition and collaboration of different concerns, and expose the oceanographer to the capabilities of the *Instrument* throughout the hierarchy. Thus, they facilitate a seamless communication along the levels of hierarchy without the oceanographer or *Instrument* being aware of the fact that they are communicating with entities outside of their authority domain.

The Rich Service architectural pattern enables hierarchical structuring of the stakeholders' logical roles into the cyber-infrastructure, and encapsulation of crosscutting concerns according to their individual policies. In addition, the concerns and authority domains of a stakeholder may be extended beyond the infrastructure under its direct control through business relations (e.g., contracts), such as owning an entity but having it managed by another stakeholder. Figure 5 depicts a Rich Services view of the OOI model, organized around authority domains and deployment concepts such as *Shore station*, *Research Facility*. The fractal nature of the architecture scales from the top-level view of the OOI system-of-systems down to the lower deployment levels in the various participating organizations. The figure displays multiple views of the system, illustrating specific roles and showing several stakeholders' crosscutting concerns.

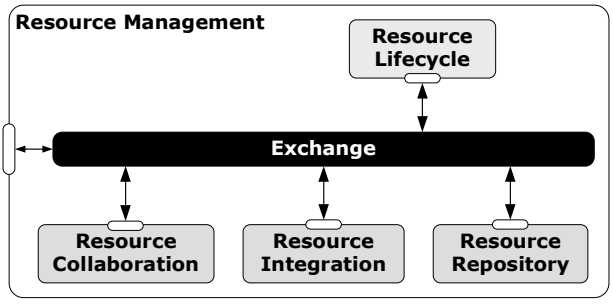


Fig. 4. OOI-CI > COI > Resource Management service decomposition

Note that at the logical level, there could be other Rich Services views of the same OOI model, organized around different concerns, such as functional subsystems.

Scenario 1. An oceanographer investigates ocean current flows near seashore and decides to retrieve the relevant data from an OOI *Archive*. This scientist belongs to a *Research Facility*, whereas the physical archive containing the data of interest is located under the authority domain of an OOI *Distribution Point*. Hence, as depicted in Figure 5, an application running under the credentials of the oceanographer has to cross multiple authority domains to retrieve the data and create the necessary plots of ocean currents. Consequently, retrieving the data is subject to the global OOI policies set by the *Observatory Management* (e.g., only public data is available), the additional policies of the *Distribution Point* (e.g., only data at least 3 months old), and finally of the *Research Facility* (e.g., only during regular business hours).

Scenario 2. Consider the following extension of the first scenario – the oceanographer needs more recent data from a remote ocean *Instrument*. Although the oceanographer owns the *Instrument*, both entities operate from different authority domains, each with its own set of requirements and policies. In more detail, the oceanographer belongs to the *Research Laboratory*, for which the core concerns are the identification and authentication of the oceanographer, and the provisioning of the research facilities within a specific set of policies regarding the available ocean instruments, knowledge bases, and experiments possible at a given time.

The identification and the management of the remote *Instrument* also concerns another stakeholder, as the *Instrument* is located deeper in the hierarchy of the *Acquisition Point*'s *Shore Station* near the seashore. Although the *Instrument* belongs to an *Science Instrument Interfacing Service* associated with the *Research Facility*, it still has to obey the set of policies regarding the power usage, allowed research activities, timing of the activities, and the available mathematical processes for the resulted data of the encompassing *Measurement Node* and the upper-level *Shore Station*. These requirements are instances of additional Rich Services provided by the stakeholders. For example, the mathematical processing services could be available in terms of CPU processing time on a supercomputer center governed by the *Observatory Management*.

For this scenario, a possible deployment plan might include classic Web services or a more general ESB-based technology, such as Mule [15]. Thus, each embedded Rich

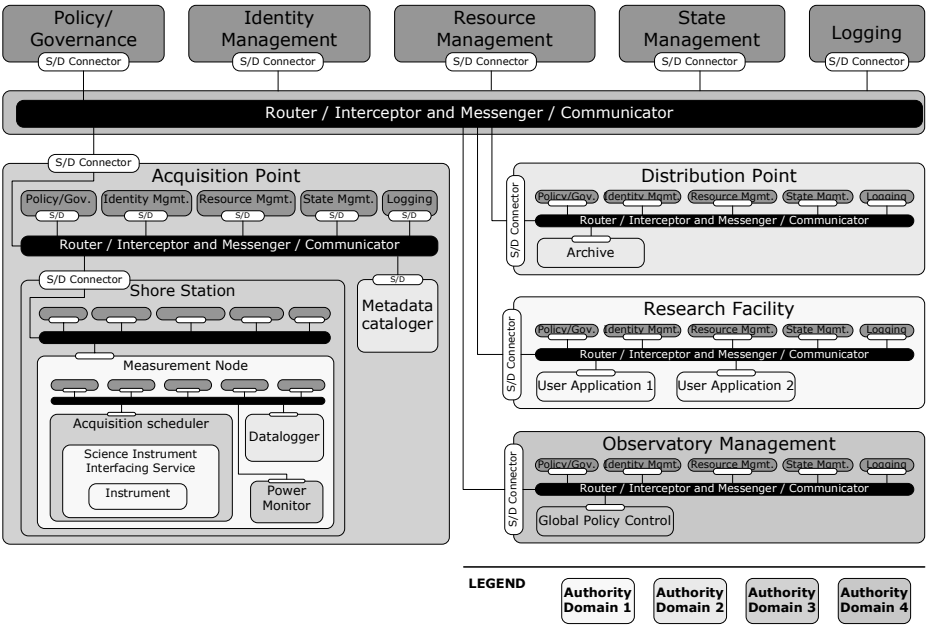


Fig. 5. Simplified deployment scenario using the Rich Services pattern

Service in the path from the oceanographer to the *Instrument* might have its own set of Rich Infrastructure Services that would transparently alter the message flow to implement identification, authentication, accounting, or logging of the data/control messages of interest at that level. At each level, one or more Rich Services might deal with the specific concerns and policies of a stakeholder. Thus, the overall role of a stakeholder in the CyberInfrastructure would be the union of all its roles on all levels of the architecture. Translated into a flat view, this union is the root of the complexity that makes the development of such systems-of-systems difficult. The vertical integration capability of the proposed architecture solves this problem and allows the decoupling of the location concerns of the stakeholders at all levels of the architecture, and allows a simplified management over the lifetime of the program.

5.1 Lessons Learned from the OOI Case Study

In addition to addressing service composition challenges, we have learnt from OOI case study a number of lessons related to ULSSIS system engineering.

Separation of concerns is key in managing system complexity. The Rich Services architectural pattern promotes the separation of business logic from infrastructure concerns. Therefore, the complexity of such a large-scale system becomes manageable by focusing on each concern separately. In the case of the OOI project, the system was decomposed into six subsystems (see Figure 2): each subsystem focuses on the services that it enables, assuming that all the infrastructure services are in place. For example, when designing the Sensing and Acquisition subsystem, the architecture

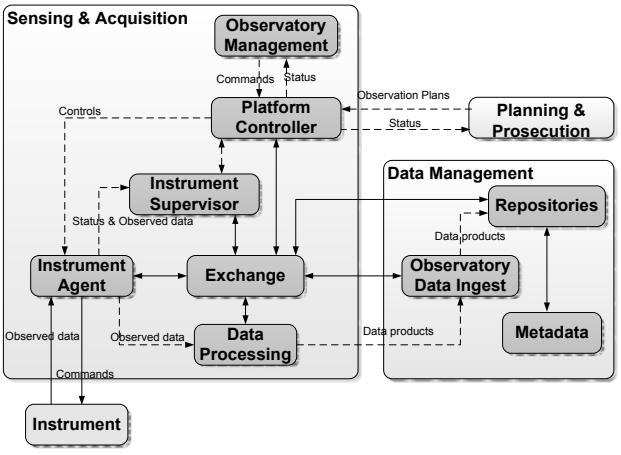


Fig. 6. Simplified view of the operational nodes (DoDAF OV-2)

team focused on concerns related to instrument control and data acquisition. Instruments can belong to a research laboratory, whereas all the deployment platforms and marine resources are under different authority domains. However, governance is handled by infrastructure services, and can be abstracted when designing the Sensing and Acquisition services.

Separation of concerns simplifies work breakdown. The responsibility for these six OOI subsystems was allocated to different teams, distributed across the U.S. Each team has as core deliverable the services of the respective subsystem; in addition, it can extend services provided by other subsystems. The Rich Services architecture provides the integration strategy that glues all services in an integrated observatory. The work-breakdown structure benefits tremendously from having infrastructure concerns addressed separately.

Presentations to stakeholders need to abstract from Rich Services details. The OOI project organized a series of workshops for gathering requirements from the community and for consolidating the design from different subsystems teams. Besides workshops, OOI undergoes a series of project reviews with the funding agency – NSF, under the management of JOI (Joint Oceanographic Institutions). All of these meetings required presentations to dozens of stakeholders from different domains. Although the audience grasps the benefits of Rich Services architecture quickly, we found out that simplified architectural diagrams and high-level depictions of the domain concepts work best. For instance, grouping all infrastructure services into the COI subsystem and presenting the other subsystems as running on top of COI is particularly adequate for domain experts who understand the science aspects of the OOI program but might be unfamiliar with service-oriented software architectures and related notations.

The OOI architecture has been developed in accordance with the Department of Defense Architecture Framework (DoDAF [15]) standard, which provides guidelines for developing architectures for large-scale systems and for presenting relevant views (i.e., all views (AV), operational views (OV), systems views (SV), and technical

views (TV)) on the architecture data in a number of products. The target audience for the OOI meetings includes decision makers, subsystem implementers, and end users. In particular, we found that the OV2 diagrams [15] – which show the operational nodes, their responsibilities, and dependencies between them – work best to convey the information flow between various system entities. For the Sensing and Acquisition subsystem, a simplified view is shown in Figure 6. The Sensing & Acquisition Services perform instrument management, mission execution, and data acquisition tasks. For simplicity, the COI subsystem is represented through its Exchange service. The dotted lines represent information flows that are implemented through the Exchange.

Difference between Rich Services and ESBs. Rich Services are useful both as a logical and deployment architecture model. ESB is a deployment technology that has the advantage of increasing code reuse due to built-in services and intercepting mechanisms. However, ESB implementations differ from the logical concepts in Rich Services, as for example, the interceptors defined in Mule v1.4 do not map directly to our interceptor abstraction. Moreover, Rich Services can be implemented in various ways (flattened vs. hierarchical deployments, ESBs vs. JVM message passing), as long as all of the logical protocols translate into the chosen implementation. Rich Services conceptualize and extend the benefits offered by ESBs at the deployment level. In particular, Rich Services are a consistent architectural blueprint that has a direct mapping to ESBs but also to other deployment technologies.

Workflows and service choreographies. Scientific communities use Data Product Generation Workflows as an automated set of processes that transform input data to output data, perform quality control, model, and visualize data. There is a good match between service-oriented architectures and workflows, as every service can become an action/step within a workflow. Therefore, workflows translate into service choreographies.

Support for Virtual Communities of Interest. Driven by the needs of stakeholders to interact to accomplish their tasks, there is an increasing desire to enable the formation of virtual communities of interest (VCOI). These VCOIs define themselves by shared computational and data resources, common agenda, communication needs, and other collaboration characteristics, and also by common policies and workflows. The OOI CyberInfrastructure not only needs to provide the core functionalities for these VCOIs to perform their tasks, but also to manage the creation, existence, dissolution of and membership in these VCOIs. This is particularly important when there is no single organization that has all the resources needed and all stakeholders in collocation: many investigators belonging to a wide range of organizations might collaborate and share resources for specific scientific experiments, thus forming a VCOI over the integration fabric provided by the OOI.

6 Related Work

Architectures. Our Rich Service architecture is an architectural style introduced in [4] that leverages well-known enterprise integration patterns [17,18,19]. Rich Services is a service-oriented architecture; it complements the OASIS SOA Reference

Model [20] and W3C Web Services Architecture [21] by providing an architectural style particularly suitable to integrating complex distributed applications such as ULSSIS. Architectural styles and patterns [22] define families of architectures whose elements are configured according to given constraints. In Rich Services, the communication is carried out by Messages exchanged over Messages Channels [19]. Services are decoupled via the Plugin pattern [18] and the Router/Interceptor as an instance of the Mediator pattern [17]. Furthermore, our Service/Data Connector uses the Façade, Proxy, and Adapter patterns [17], as well as the Messaging Gateway pattern [19]. Finally, the Composite pattern [17] enables the hierarchical composition of Rich Services.

Rich Service architecture addresses key integration and extensibility requirements, while providing a systematic way to model various stakeholders' crosscutting concerns. As noted in [23], software architectures define the configuration of architectural elements. [24] argues that to provide adaptability for ULS, we need to pay particular attention to couple software artifacts and processes with the stakeholders and environments that influence the decisions to which systems must adapt. Thus, [24] suggests to scope architectures as ontology of decision makers, design decisions, and constraints on subsequent design decisions. Our approach is to use a service-oriented development process [13] that encompasses activities from use case elicitation, service definition, mapping to Rich Services, through physical network deployment. With the Rich Services architecture, we see a service composition as the binding of participating services in respect not only to the behavior of the services, but also to their requirements and policies. Our process is iterative and supports partial requirements and partial specifications of the system. For example, at the stage of creating Rich Services, it is common to discover additional opportunities for crosscutting concerns such as QoS monitoring and failure management. These concerns reflect functional and non-functional facets of requirements, which may generate additional use cases resulting in spiraling back to the initial development stages.

The elaboration of a formal model for service interfaces and their composition is beyond the scope of this paper. However, the work in [25] is a valid foundation for a formal underpinning for Rich Services; it introduces the concepts of components and services as total and partial behaviors, and uses Message Sequence Charts (MSCs) for the definition and composition of services.

Web Services composition. The Rich Services architecture supports the flexible composition of services. Service composition has been one of the most active development areas for Web service technologies in recent years. Two interesting directions for Web service composition are business workflows and the semantic Web.

The business world views Web services as interfaces to business processes and represents service composition as a workflow: Web services are specified in WSDL, and the flow must is explicitly specified in a flow-composition language for orchestration (e.g., BPEL [26]) or choreography (e.g., WS-CDL [27]). Rich Services support these composition approaches – their encapsulation and hierarchy guide developers to focus on one hierarchical level at a time.

The Semantic Web [28] community uses semantic representations of services using languages such OWL-S [29] to address the challenging problem of flexible runtime discovery, binding, and automatic composition of services. The flexible architecture

of Rich Services and the ESB deployment strategy allow us to expose richer interfaces and reconfigure communication at run time, thereby enabling the use of ontology-based composition techniques.

Extensibility and Adaptability for ULSSIS. Already in [30] Parnas stresses the importance of designing software in a way that can be tailored to the needs of different users. For ULSSIS, this is of utmost importance. We make the recommendation of designing for change a central element of the Rich Service architecture. Rich Services are decoupled via the Router/Interceptor layer and the Service/Data Connector. Crosscutting concerns are handled separated via dynamic behavior injection. Thus, emerging requirements can often be implemented with minimal or no changes to working code. Instead, changes can be implemented by intercepting and transforming message flows.

[31] emphasizes the need for adaptations in ULSSIS and proposes to use techniques from autonomic computing [32], which refers to any system that manages itself based on high-level objectives and achieves reconfiguration, self-optimization, self-healing, and self-protection. Monitoring is an important aspect of self-managing systems. In general, monitoring is difficult [33] because, among others, it may impact ULSSIS performance, an application might have heterogeneous monitoring techniques from different parties, and the set of techniques might change over time. In Rich Services, monitoring is a crosscutting concern that is handled by an infrastructure service; different monitoring techniques can be used at each level in the hierarchy. The system can easily handle changes in the monitoring services, because such changes do not affect the actual application services.

[31] investigates design patterns for self-monitoring systems, such as Content-based routing, Look-up Table, Stream Splitter, Stream Merger, etc. Rich Services also use service registry, enable dynamic interception and routing of messages, and the Service/Data Connector acts as a gateway between different Rich Services levels. [31] builds upon results from frameworks such as [34] and [35] for monitoring distributed applications, and uses the model-based development process from [36], which separates the adaptation behavior and non-adaptive behavior specifications of adaptive programs, thus making the models easier to specify and amenable to automated analysis and visual inspection. We also emphasize the need for separation of concerns in ULSSIS systems. For example, we see failure management as a crosscutting quality concern that requires placing the interactions between components in the center of attention to transition from a per-component basis to an end-to-end failure management notion in ULSSIS systems. We have introduced a service-oriented development approach [14] that addresses both component- and service-level failures, and establishes a clean separation between the services provided by the system, the failure models, and the architecture implementing the services. By exploiting the relationship between service specifications and failure management specifications, we were able to formally verify (model check) the fail-safety of a system under a given failure hypothesis.

Development processes for ULSSIS. Traditional approaches to system engineering define clear boundaries between a system and its environment and define a software engineering development process that matches clear requirements to specific components in the architecture. In systems of systems (SoS), individual systems are no longer seen as bounded entities, but rather interacting with other participants to form a larger system based on end-to-end business processes and requirements [37]. Our

Rich Services development process [13] fits well with large-scale SoS integration recognizing that system requirements are generally not defined in terms of architectural components; instead, they typically span across the various components of the system, establishing complex interaction dependencies. Therefore, we place services in the center of the attention: we view services as interaction patterns among the system entities involved in establishing a particular piece of functionality. This service-oriented development process also achieves a clean separation of the logical model of the system and its implementation. The process can be integrated with SoS development processes such as SoSE [38] and ICM [39], which address the issue of responsibilities and coordination between the integrated SoS and the constituent subsystems. Furthermore, cost estimation models and tools [40] support estimating the effort for the development and evolution of SoS.

In the SoSE model (systems-of-systems engineering model) [38] the component systems retain their responsibility for management and engineering, and SoS managers and engineers do not control but collaborate with systems managers and engineers to influence the systems developments such as to address SoS objectives. The model encourages loose coupling between systems to support adaptability during system evolution.

The Incremental Commitment Model (ICM) [39] is a risk-driven framework that combines agile and plan-driven processes, and emphasizes continuous verification and validation. ICM has iterative development cycles focusing on incremental growth of system definition and stakeholder commitment and satisfaction. For systems of systems, ICM implies to coordinate the activities for each system using the Life Cycle Objectives (LCO), Life Cycle Architecture (LCA), and Operational Capability Release (OCR) reviews. The OOI project integrates this model with Rich Services, and is currently in the Life Cycle Objectives phase for its subsystems.

7 Conclusion

We presented a set of challenging requirements for designing, building, and operating ultra large-scale software-intensive systems, using the OOI-CI as running example. When using SOAs for modeling such systems, these requirements translate into corresponding challenging service composition requirements. We are successfully using the Rich Services framework within OOI-CI to answer these challenges. Rich Service is an extension of the standard service notion based on an architectural pattern that allows hierarchical decomposition of system architecture according to separate concerns. The Rich Services communication infrastructure enables loose coupling and seamless communication between services. Such capability drives the evolution of the system by providing the underlying mechanisms to handle changes, dynamic reconfiguration, and policy enforcement.

While the Rich Service blueprint covers a significant part of the requirements spectrum for ULSSIS, creating a comprehensive engineering approach to service-oriented ultra-large-scale systems requires a long-term research endeavor addressing theory to modeling to deployment. The focus shifts from the production of components (open or monolithic) to the choreography of components or the services they offer; hence, our approach with Rich Services pioneers many future research directions.

Acknowledgements

We are grateful to our colleagues on the OOI-CI project, specifically John Orcutt, Frank Vernon, Matthew Arrott, Alan Chave, Oscar Schofield, Cheryl Peach, Jack Kleinert, Von Welch, Munindar Singh, and Michael Meisinger whose expertise has contributed to the case study presented here. We are also grateful to our colleagues from the other OOI implementing organizations, as well as the sponsors from NSF and JOI for their comments on the OOI-CI design. Our work was partially supported by NSF within the projects "ASOSA" (CCF #0702791) and "OOI-CI", as well as by funds from the California Institute for Telecommunications and Information Technology (Calit2).

References

1. Brooks, F.P.: The Mythical Man Month (Anniversary Edition). In: No Silver Bullet - Essence and Accident, ch. 16. Addison Wesley, Reading (1995)
2. Northrop, L., Feiler, P., Gabriel, R.P., Goodenough, J., Linger, R., Longstaff, T., Kazman, R., Klein, M., Schmidt, D., Sullivan, K., Wallnau, K.: Ultra-Large-Scale Systems: The Software Challenge of the Future. Software Engineering Institute. Carnegie Mellon University, Pittsburgh (2006)
3. Ocean Observatories Initiative – CyberInfrastructure (OOI-CI) funded by the NSF, http://www.oceanobservatories.org/
4. Arrott, M., Demchak, B., Ermagan, V., Farcas, C., Farcas, E., Krueger, I.H., Menarini, M.: Rich Services: The integration piece of the SOA puzzle. In: Proceedings of the IEEE International Conference on Web Services (ICWS), pp. 176–183. IEEE Press, Los Alamitos (2007)
5. Boehm, Turner, R.: Balancing Agility and Discipline: a Guide for the Perplexed. Addison-Wesley Longman Publishing Co., Inc., Amsterdam (2003)
6. OASIS SOA Reference Model (2008), http://www.oasis-open.org/committees/soa-rm/
7. W3C Web Services (2002), http://www.w3.org/2002/ws/
8. Marca, D.A., McGowan, C.L.: SADT: Structured Analysis and Design Technique. McGraw-Hill, Inc., New York (1987)
9. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W.: Object-Oriented Modeling and Design. Prentice-Hall, Inc., Englewood Cliffs (1991)
10. Object Management Group: The Common Object Request Broker: Architecture and Specification (1991), http://www.omg.org/technology/documents/spec_catalog.htm
11. Ermagan, V., Krüger, I.H., Menarini, M.: A Fault Tolerance Approach for Enterprise Applications. In: Proceedings of the IEEE International Conference on Services Computing, SCC (2008)
12. Ermagan, V., Farcas, C., Farcas, E., Krueger, I.H., Menarini, M.: A service-oriented blueprint for COTS integration: the hidden part of the iceberg. In: Proceedings of the ICSE Second International Workshop on Incorporating COTS Software into Software Systems: Tools and Techniques (IWICSS), p. 10. IEEE Computer Society Press, Los Alamitos (2007)

13. Demchak, B., Farcas, C., Farcas, E., Krueger, I.H.: The treasure map for Rich Services. In: Proceedings of the 2007 IEEE International Conference on Information Reuse and Integration (IRI), pp. 400–405. IEEE, Los Alamitos (2007)
14. Ermagan, V., Krueger, I., Menarini, M., Mizutani, J.I., Oguchi, K., Weir, D.: Towards model-based failure-management for automotive software. In: Proceedings of the ICSE Fourth International Workshop on Software Engineering for Automotive Systems (SEAS), p. 8. IEEE Computer Society Press, Los Alamitos (2007)
15. Mule Open Source ESB and Integration Platform, http://mule.mulesource.org/wiki/display/MULE/Home
16. DoD Architecture Framework Version 1.5 (April 2007), http://www.defenselink.mil/cio-nii/docs/DoDAF_Volume_II.pdf
17. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, Reading (1995)
18. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley Professional, Reading (2003)
19. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Professional, Reading (2004)
20. MacKenzie, C., Laskey, K., McCabe, F., Brown, P., Metz, R.: Reference Model for Service Oriented Architecture 1.0. OASIS (October 2006), http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf
21. Booth, D., et al.: Web Services Architecture. W3C (February 2004), http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/
22. Monroe, R., Kompanek, A., Melton, R., Garlan, D.: Architectural Styles, Design Patterns, and Objects. IEEE Sofware 14, 43–52 (1997)
23. Perry, D., Wolf, A.: Foundations for the Study of Software Architecture. SIGSOFT Software Engineering Notes 17(4), 40–52 (1992)
24. Song, Y., Rai, D., Sullivan, K.: Adaptation architectures cross levels. In: Proceedings of the 2nd international Workshop on Ultra-Large-Scale Software-intensive Systems, ULSSIS 2008, Leipzig, Germany, May 10 - 11, pp. 27–28. ACM, New York (2008)
25. Broy, M., Krueger, I.H., Meisinger, M.: A Formal Model of Services. ACM Transactions on Software Engineering Methodology 16(1), 5 (2007)
26. Andrews, T., et al.: Business Process Execution Language for Web Services (BPEL4WS) 1.1 (May 2003), http://www.ibm.com/developerworks/webservices/library/ws-bpel
27. Kavantzas, N.: Web Services Choreography Description Language Version 1.0. W3C (November 2005), http://www.w3.org/TR/2005/CR-ws-cdl-10-20051109/
28. Berners-Lee, T., Hendler, J., Lassila, O.: The Semantic Web. Scientific American 284(5), 34–43 (2001)
29. Martin, D., et al.: OWL-S: Semantic Markup for Web Services. W3C (November 2004), http://www.w3.org/Submission/2004/SUBM-OWL-S-20041122/
30. Parnas, D.: Designing Software for Ease of Extension and Contraction. IEEE Transaction on Software Engineering SE-5, 128–138 (1979)
31. Ramirez, A.J., Cheng, B.H.C.: Design patterns for monitoring adaptive ULS systems. In: Proceedings of the 2nd international Workshop on Ultra-Large-Scale Software-intensive Systems, ULSSIS 2008, Leipzig, Germany, May 10 - 11, pp. 69–72. ACM, New York (2008)
32. Kephart, J., Chess, D.: The vision of autonomic computing. IEEE Computer 36(41-50) (2003)

33. Garlan, D., Schmerl, B., Chang, J.: Using gauges for architecture-based monitoring and adaptation. In: Working Conference on Complex and Dynamic Systems Architecture, Brisbane, Australia (2001)
34. Andreozzi, S., Bortoli, N.D., Fantinel, S., Ghiselli, A., Rubini, G.L., Tortone, G., Vistoli, M.C.: Gridice: a monitoring service for grid systems. Future Gener. Comput. Syst. 21(4), 559–571 (2005)
35. Huang, A.-C., Garlan, D., Schmerl, B.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. In: ICAC 2004: Proceedings of the First International Conference on Autonomic Computing. IEEE Computer Society, Los Alamitos (2004)
36. Zhang, J., Cheng, B.H.C.: Model-based development of dynamically adaptive software. In: ICSE 2006: Proceeding of the 28th international conference on Software engineering, pp. 371–380. ACM Press, New York (2006)
37. Lane, J.A., Dahmann, J.S.: Process evolution to support system of systems engineering. In: Proceedings of the 2nd international Workshop on Ultra-Large-Scale Software-intensive Systems, ULSSIS 2008, Leipzig, Germany, May 10 - 11, pp. 11–14. ACM, New York (2008)
38. Dahmann, J., Lane, J., Rebovich, G., Baldwin, K.: A Model of Systems Engineering in a System of Systems Context. In: Proceedings of the Conference on Systems Engineering Research, Los Angeles, CA, USA (April 2008)
39. Boehm, B., Lane, J.: Using the Incremental Commitment Model to Integrate System Acquisition. Systems Engineering, and Software Engineering 19(10), 4–9 (2007) (CrossTalk)
40. Lane, J., Boehm, B.: Modern Tools to Support DoD Software Intensive System of Systems Cost Estimation. In: Data and Analysis Center for Software, Rome, NY (2008)
41. Schmidt, D.C., Stal, M., Rohnert, H., Buschmann, F.: Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, vol. 2. Wiley, Chichester (2000)

On the Pragmatics of Model-Based Design

Hauke Fuhrmann and Reinhard von Hanxleden

Real-Time and Embedded Systems Group, Department of Computer Science
Christian-Albrechts-Universität zu Kiel, Olshausenstr. 40, 24118 Kiel, Germany
{haf,rvh}@informatik.uni-kiel.de
www.informatik.uni-kiel.de/rtsys/

Abstract. The pragmatics of model-based design refers to the practical aspects of handling graphical system models. This encompasses a range of activities, such as editing, browsing or simulating models. We believe that the pragmatics of modeling deserves more attention than is has received so far. We also believe that there is the potential for significant productivity enhancements, using technology that is largely already available. A key enabler here is the capability to automatically and quickly compute the layout of a graphical model, which frees the designer from the burden of manual drawing. This capability also allows to compute customized view of a model on the fly, which offers new possibilities for interactive browsing and for simulation.

1 Introduction

Linguists distinguish the syntax, semantics and pragmatics of languages. Together these three categories are referred to as *semiotics*—the study of how meaning is constructed and understood. All three categories can be applied to programming languages as well as natural languages. In the context of programming languages, *syntax* is determined by formal rules saying how to construct expressions of the language, *semantics* determines the meaning of syntactic constructs, and the *pragmatics* of a language refers to practical aspects of how constructs and features of a language may be used to achieve various objectives [1]. In this paper, we argue that the pragmatics of modeling languages deserves more attention than it has received so far. Specifically, it appears that the practical issues of how to create, maintain, browse and visualize effective graphical models have been neglected in the past. This largely offsets the inherent advantages of visual languages, makes it difficult to design complex systems, and unduly limits designers' productivity. Petre [2] quotes a professional developer as follows: "I quite often spend an hour or two just moving boxes and wires around, with no change in functionality, to make it that much more comprehensible when I come back to it."

Traditionally, "pragmatics" refers to how elements of a language should be used, e. g., for what purposes an assignment statement should be used, or under what circumstances a level of hierarchy should be introduced in a model. It is usually not considered how the practical design activities themselves (editing,

C. Choppy and O. Sokolsky (Eds.): Monterey Workshop 2008, LNCS 6028, pp. 116–140, 2010.

browsing, etc.) are performed—simply because this is usually not much of an issue when textual languages are concerned. There may be differences in convenience of use in different text editors, and integrated design environments (IDEs) can provide various levels of support in building and maintaining large software artifacts. However, the basic mechanics of writing or changing a line of code is rather standard and efficient. In comparison, the mechanics of editing a graphical model are much more involved, and it appears that there is much to be gained in this area. Hence by "pragmatics of modeling languages" we here slightly extend the traditional interpretation of "pragmatics" to encompass all practical aspects of handling a model in a model-based design flow, including the traditional aspect of how a model should be constructed to effectively communicate its meaning.

There are several established fields that can provide valuable input here, such as the area of human computer interaction, cognitive psychology, and the graphical layout community. For example, there are fundamental practical differences in using textual or graphical languages [1], and freeing the modeler from the burden of manually drawing a graphical model opens the door to a number of productivity-enhancing techniques that allow to combine the best of both worlds [3]. Furthermore, there are already a number of paradigms well established in software engineering that could be put to use for model-based design processes, including the design of the modeling infrastructure itself. For example, the state of the practice in creating a graphical model, say, a dataflow diagram or a Statechart, is to directly construct its visual representation with a drag-and-drop (DND) What-You-See-Is-What-You-Get (WYSIWYG) editor, and henceforth rely on this one representation. We here propose instead to apply the Model-View-Controller (MVC) paradigm [4] to separate a model from its representation (view). Together with a modeling environment (the controller/editor) capable of automatic model layout, one can thus provide flexible representations. These views can be adapted according to specific design activities, balancing useful information with cognitive complexity [5].

In this paper, we survey the different aspects of the pragmatics of graphical modeling languages. This covers a broad range of existing work, as well as a number of observations and proposals that to our knowledge have not been reported on before. As space is limited, we do not attempt to investigate any of these aspects in much detail here, but rather try to cover as much ground as possible. A non-trivial question at the onset was how to organize the subject matter. There exist extensive surveys in the area of model-based design, see for example Estefan's overview of model-based systems engineering methodologies [6], or the overview of hybrid system design given by Carloni et al. [7]. An annotated bibliography by Prochnow et al. [8] inspects the visualization of complex reactive systems. There exist numerous surveys on automatic graph drawing, which we consider an essential enabler for efficient modeling [9,10]. However, we are not aware of an existing taxonomy that focuses on the aspect of pragmatics. We here opt for the aforementioned MVC concept as a guiding principle. For a first overview, see Fig. 1. In some cases, it may be arguable how a certain aspect

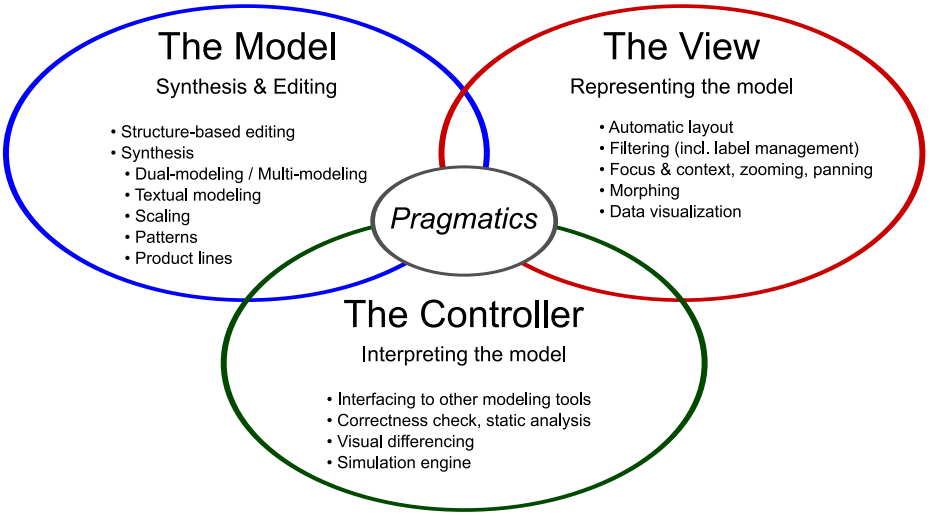


Fig. 1. The MVC paradigm applied to the pragmatics of graphical model-based system design

should be classified; e. g., we here consider editing to be part of the model, but it could also be classified as part of the controller. However, we still find the MVC classification helpful.

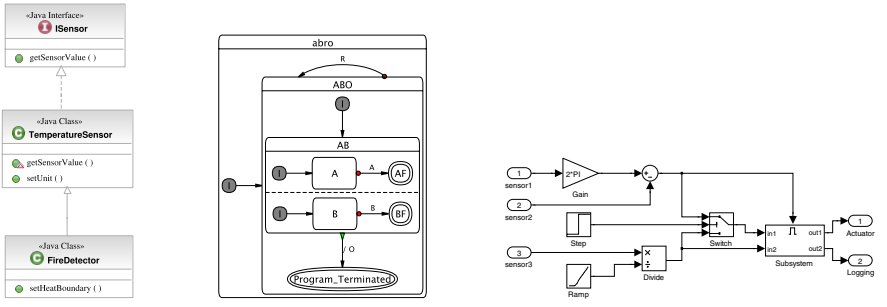
This structure is also reflected in the organization of this paper, except that we start with the view (Sec. 2), followed by the model (Sec. 3) and the controller (Sec. 4). We conclude in Sec. 5.

Example figures in the following sections are mainly taken from different graphical modeling tools like Mathwork's Matlab/Simulink, Esterel Technology's E-Studio and SCADE and graphical editors basing on the Eclipse platform. None of the tools handles pragmatics very well so far, so the images are mainly for illustration of the concepts but not for showing the state-of-the-art in implementation. An implementation of the concepts presented in this paper is ongoing work in the project Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER)[1].

2 The View—Representing the Model

We believe that a key enabler for efficient model handling is the capability to automatically compute the layout of a graphical model. If one frees the user from the burden of manually setting the coordinates of nodes and bendpoints, sizes of boxes and positions of connection anchor points, this can open up enormous potentials. The following section explores this further.

[1] http://www.informatik.uni-kiel.de/rtsys/kieler



(a) UML Class Diagram (b) Statecharts (here SyncCharts) (c) Dataflow diagrams (here Matlab/Simulink)

Fig. 2. Different graphical syntaxes with different properties for their layout

2.1 Automatic Layout

The correct use of pragmatic features, such as layout in graph-based notations, is a significant contributory factor to the effectiveness of these representations [2]. Automatic layout has to be appealing to the user such that he or she is willing to replace optimized manual layout with an automatically created one. Additionally this layout capability would have to be deeply integrated into the modeling tool and optimized for the respective graphical language syntaxes.

One must recognize that at this point, the automatic layouting capabilities offered by modeling tools, if they do offer any capabilities at all, tend to be not very satisfying. A major obstacle is the complexity and unclarity of this task. What are adequate aesthetic criteria for "appealing" diagrams [11,12,13]? Are there optimal solution algorithms or heuristics with acceptable results that adhere to the desired aesthetic criteria? An important aspect is the usage of *secondary notation*, which is specific to the modeling language used [14]. Used properly, an automatic layout does not only provide aesthetically pleasing diagrams, but can also give the viewer valuable cues on the structure of a model. For example, a standardized way of placing transition labels (e. g., "to the left in direction of flow") can solve the often difficult label/transition matching problem. Similarly a standardized direction of flow (e. g., "clock wise") can give a quick overview of the flow of information, without having to trace the direction of individual connections.

Fig. 2 shows three examples of different graphical formalisms that pose different layout challenges. Unified Modeling Language (UML) Class Diagrams look quite close to the standard graph layout problem, although sometimes hierarchy might be added by displaying packages in the diagram. While usual relations can be regarded as any graph edges, inheritance relations as shown in Fig. 2(a) have a special role. They are typically drawn from top to bottom, which is a strong constraint for the layout algorithm. So even here one needs a specialized layout algorithm for this diagram type [15].

Statemachines fit pretty well to graph layout, but introducing hierarchy requires special handling. In a diagram with hierarchy and without any inter-level connections crossing hierarchy boundaries, the layout algorithm for a flat layout can be called recursively. This was employed for Statecharts as shown in Fig. 2(b) using the layered based Sugiyama layouter of the GraphViz library in the KIEL project [3]. Small enhancements of the graphical syntax might have severe consequences for the layout. Inter-level transitions, which are possible in some Statechart dialects as UML State Machine Diagrams or Stateflow of Matlab/Simulink, cannot be layouted with this approach and would require a special handling again.

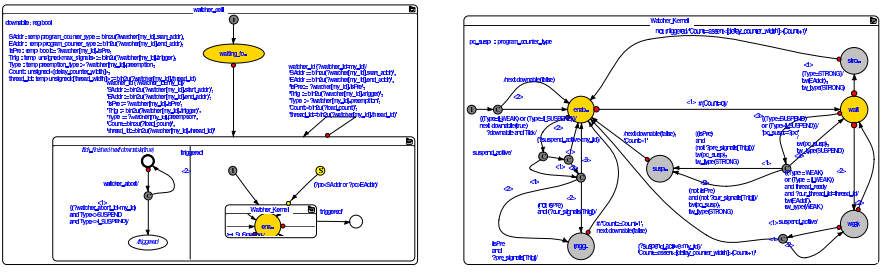
Another special class are actor oriented dataflow languages [16]. The notion *dataflow* sometimes is used in different contexts resulting in different diagram syntaxes. We here consider languages usually used in the control engineering domain such as Ptolemy, the Safety Critical Application Development Environment (SCADE), or Matlab/Simulink (Fig. 2(c)). The connections denote flows of data and two distinct connections will likely carry different data and possibly different data types. Data are consumed by operators, and to distinguish the different incoming and outgoing data sources and sinks, an operator has special input and output *ports*. For many operators it is very important to specify explicitly which data flow is connected to which port because an alternation would also alternate the semantics. The example shows subtraction, division and switch operators which are not commutative and hence need their incoming flows exactly at the right input ports. The graphical representation also reflects this issue by presenting specific anchor points for the connections at the border of the operators. For the mentioned languages these ports have fixed positions relative to the operator, usually showing the data flow from left to right by positioning inputs left and outputs right. However, some special purpose ports may also be positioned on top or bottom of the operator, in general at pre-defined and static locations. These *port constraints* induce a great complexity to the problem and require special care such as by the approaches of Eiglsperger et al. [17] or a modified Sugiyama layout as implemented in the KIELER project[2].

Summing this up, we cannot hope for one ultimate layout algorithm that is applicable for all languages and applications. Instead, we need a set of different layouters to cover a wide range of language syntaxes and layout styles.

2.2 Filtering

Card et al. [18] define approaches for reducing information in a diagram: *Filtering*, *Selective Aggregation*, *Highlighting* and *Distortion*. A filter simply hides a set of objects in the diagram, to reduce the complexity of a diagram. For technical scalability issues it is often not feasible to construct and inspect models with many objects—hundreds or thousands of nodes—but consistently working with filters it can be. Only a small set of objects should be visible while all others are hidden and do not consume graphical system resources. By navigating through

[2] http://www.informatik.uni-kiel.de/rtsys/kieler/



(a) Top-level state (b) Inner state

Fig. 3. Filtering in E-Studio, showing a part from a processor design [19]: The composite state in the lower right corner of (a) displays only a very small part of its inner life (b)

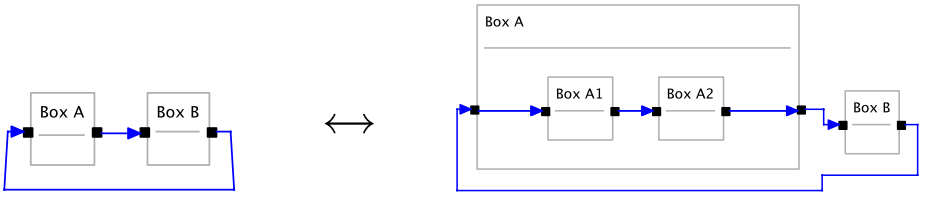
the model, a user reveals some parts and hides others. In order to properly work, we need strategies to apply this automatically to free the user of the burden to manually selecting the items to show and to hide. (This also leads to the focus and context paradigm, see Sec. 2.4.)

Simple filters can already be found in some tools that hide objects on the canvas while the canvas size resp. the bounding box stays the same size. Hence only the number of elements is reduced but not the size and therefore the same zoom level or paper size is required to display the model and there is hardly any chance to see more of the surrounding context as before. A rather unusual way of filtering can be used in Esterel Studio, see Fig. 3. The hierarchy mechanism in E-Studio allows to create the relatively clearly arranged top-level diagram Fig. 3(a). However, the macrostate Watcher Kernel in the lower right reveals only a very small part of its contents, the rest is hidden. One has to manually open the Watcher Kernel state in a new canvas in order to see its whole extend shown in Fig. 3(b) where a complex inner life is revealed compared to what small part of it is shown in the parent. This feature becomes more useful in combination with automatic layout that uses the free space gained from the filters.

Dynamic Visible Hierarchy. Dynamic hierarchy is a special case of a filter where all children of some parent object are filtered. For filters one might select to hide items regardless of the hierarchy level to reduce the complexity, see Fig. 4 for a simple example. This corresponds to the *folding features* of text or XML editors [20].

2.3 Label Management

Working with real-world applications quickly leads to the question of how to handle long labels. Label placement is a big issue in graph drawing [21] and geography with



(a) Composite Box A folded

(b) Composite Box A unfolded

Fig. 4. Example for dynamic visible hierarchy, here for an actor oriented data-flow language implemented in KIELER. This utilizes collapsible compartments and a layer-based automatic layout algorithm supporting port constraints.

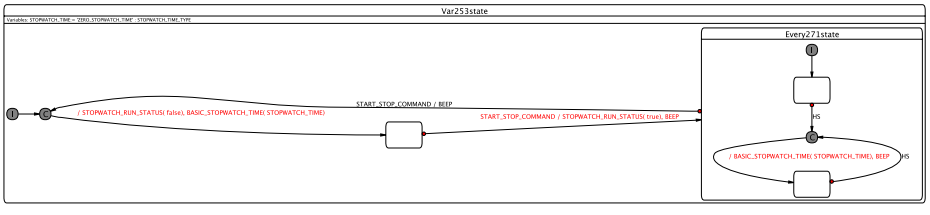


Fig. 5. Long labels prevent good layout. Here a small part of Harel's wristwatch example [25], converted from Esterel to Statecharts [26].

map feature labeling [22]. The problem is computationally intensive, for bended edges it is NP-hard [23]. In map labeling labels are rather short—city, street or river names—but in arbitrary Domain Specific Languages (DSLs) they do not need to be, as Fig. 3(a) shows. We assume to have an automatic layout algorithm that takes care of the label positioning, taking the label as is and not changing it. There are innovative approaches changing the diagram syntax, e. g. to replace an edge by the label itself [24] by optical scaled down distortion. However, we do not consider such invasive changes as universal option.

Instead, we try to dynamically reduce the complexity of the label to give the layouter better chances to find appealing layouts and to avoid difficulties as illustrated in Fig. 5. A *label filter* might use different strategies, see also Table 1. *Wrapping* aims to compact the label by wrapping the text while *abbreviation* hides part of the text to actually shorten the length of the string. Syntactically arbitrary labels might be handled with possibly suboptimal results. Even soft wrapping respecting identifiers will wrap compound labels inappropriately by not keeping related identifiers on one line. Semantical abbreviation could hide specific token types while showing only more important ones, like operators vs. variable/signal references. With a *label manager* in charge, the labels can be dynamically displayed with different levels of detail.

Table 1. Ways to reduce label complexity temporarily. Here for a Statechart transition label

Original	`(not SignalA) and (not SignalB) / SignalC(counter)`		
Wrapped	syntactical	hard	`(not SignalA) and (not SignalB) / Signal C(counter)`
		soft	`(not SignalA) and (not SignalB) / SignalC (counter)`
	semantical		`(not SignalA) and (not SignalB) / SignalC(counter)`
Abbreviated	syntactical		`(not SignalA) and (not Si...`
	semantical		`SignalA, SignalB / SignalC`

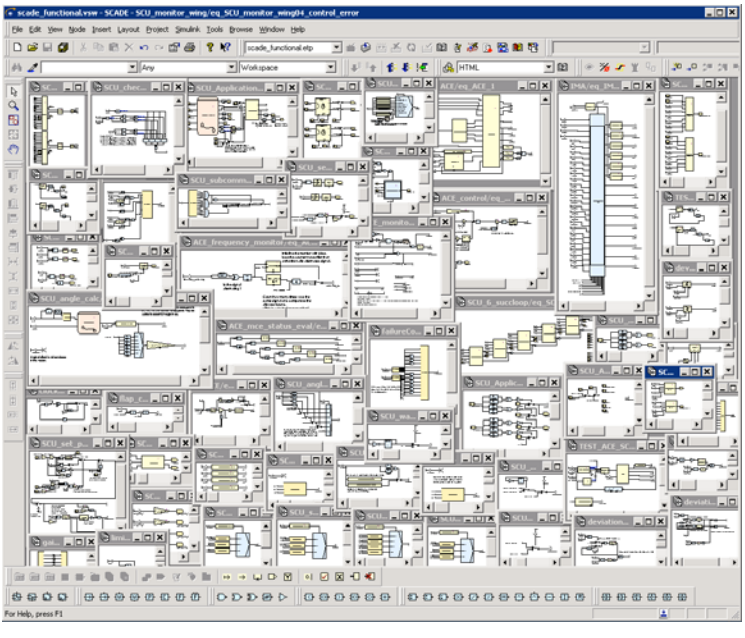
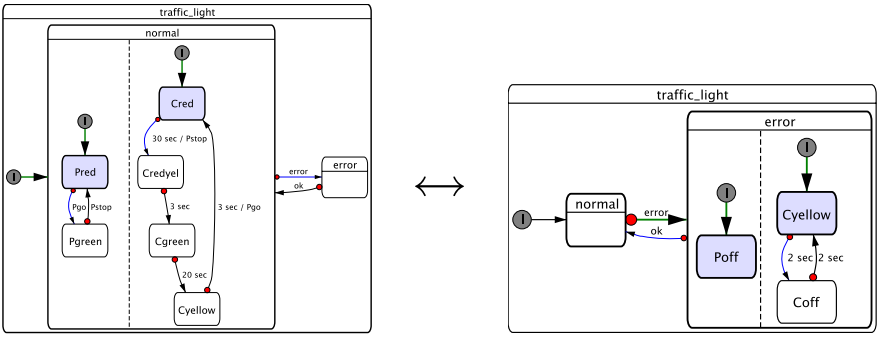


Fig. 6. Illustration for the lack of proper view management: showing the whole system entails loosing details, windows get to small to be usable

2.4 Focus and Context

In classical modeling environments, the user typically has the alternatives of either seeing the whole model without any detail, or seeing just selected parts of the model. Fig. 6, from an avionics application, shows what may happen if one does try to see the whole system. To find a way out of this dilemma, we



(a) View with focus on state normal (b) View with focus on state error

Fig. 7. Semantical graphical focus and context in KIEL: The two large composite states normal (a) and error (b) are only displayed in full detail when the respective state is active. The inactive state is filtered by dynamic hierarchy and forms the context.

note that when working with a model, it is common that there are parts of the model of particular interest for the current operation or analysis, which we refer to as the *focus*. Other objects next to it comprise the *context*, which might be important information to understand the focus objects but may be displayed with less details. This leads to a *focus and context* approach where filters are employed to hide irrelevant objects [27].

Focus and Context in KIEL. The Kiel Integrated Environment for Layout (KIEL) project [3] uses a semantical graphical focus and context technique to hide details in the context while highlighting the focus. It is *semantical*, because the decision of objects to be filtered is made automatically from semantic background information from the model [27]. The concept is used during simulation of a Statechart diagram where the focus seems to be quite natural to state machines: the currently active states. Hence all active simple states are displayed together with their whole hierarchy, i. e. all ancestor states (which actually are also active). Dynamic visible hierarchy, as presented in Sec. 2.2, is used to show all sibling states but hide their contents. Hence, tidy diagrams are presented always with reduced complexity as shown in Fig. 7.

During simulation the user never sees the whole diagram but only either one of the focused views. Smooth animated morphs between the views guide the mind of the user from one view to the other so he or she can keep the *mental map* of the whole application [28].

Alternatives for Focus and Context. Experience showed that KIEL's specific interpretation of what objects comprise the focus and which the context is not always optimal. Sometimes it is difficult to follow the reasons of the view change, i. e. the switch from state normal to error. Signals emitted in the collapsed state can cause this change but are immediately hidden and hence the user

cannot follow the causal event chain. This calls for a more general approach for applying focus and context techniques. Even for this specific DSL one can come up with various other schemes to select the focus objects.

- One could show an intermediate step between the transition where the former active and the new active states are focused both.
- One might decide the focus by active transitions instead of states—this could also filter parallel regions that do not change configuration.
- The context does not necessarily must go up to the top level, but might be limited to some number of hierarchy levels.
- *Meta focus:* one could specify a more abstract focus, e. g. "focus on signal S," which would set the concrete focus on transitions/states that reference S.

2.5 View Management

Considering diagram types other than Statecharts, it is not that obvious how to select the focus of the diagram, because there might be no such thing as an active state—e. g. in dataflow diagrams sometimes all operators are active in every step—or there is no visible step-wise simulation at all—e. g. structural diagrams such as UML class diagrams. To broaden applicability, it appears natural to upgrade layout information and directives to "first-class-citizens." By this we mean that the view of a model becomes part of the state of a model, which can be controlled by the user, the modeling tool, or the model itself. An engine for *view management* could for example categorize graphical entities in focus and context, maybe even multiple levels of context by setting different *levels of detail* as denoted by Musial for UML [29]. These and other aspects of view management are depicted in Fig. 8.

The view manager needs to listen to *triggers*, or *events*, at which it might change between the dynamic views, showing the user some objects in the focus and others in the context. These triggers might be *user triggers*, induced manually by the user, e. g. manually clicking on fold/unfold buttons at parent nodes or manually changing the focus by selecting a different node. They could also be *system triggers*, produced by the machine by some automatic analysis, semantical information, progress of time (real or logical), etc. *Memorized triggers* can for example be trigger annotations stored persistently with a model.

Obviously, this view manager can hardly be one monolithic application that carries all information and is applicable for all types of DSLs and application environments. We need a way to efficiently specify both the *triggers* to listen to and the *effects* that shall be performed. This *view management scheme* (VMS) needs to be provided by the developer, either by the application developer for application specific schemes or by the tool creator for more general schemes applicable for a whole DSL. For a practical user interface this VMS should be expressed by a simple syntax, maybe close to some general purpose scripting language. It would require expressions to

1. address different user triggers (mouse clicking, keyboard events),
2. specify custom system triggers,

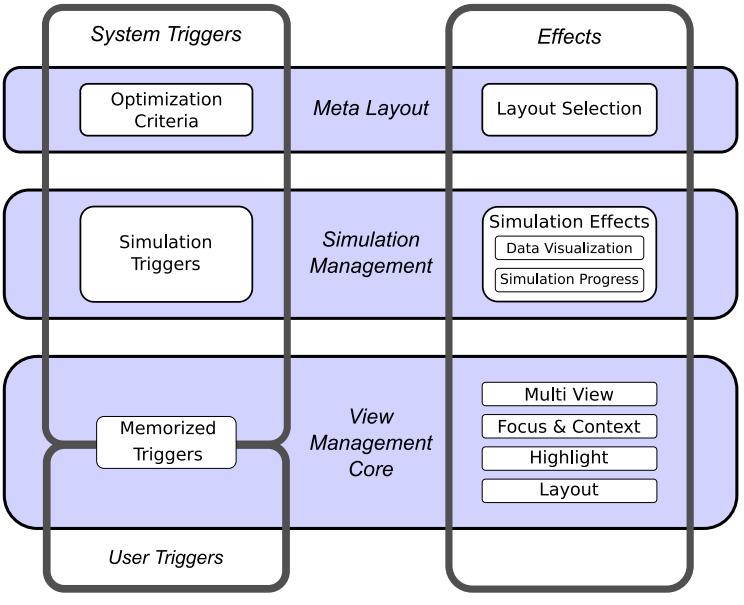


Fig. 8. Aspects of view management

3. address different system triggers,
4. address different visualization effects (folding, unfolding, filtering, layout triggering, choose layout algorithms), and
5. address graphical diagram objects or their properties, either specific objects (e. g. "State A") or classes of objects (e. g. "a node of type *state*") or specific *patterns* of such objects.

Some of the items can be implemented using standard techniques, such as addressing model elements. A set of predefined user triggers and visualization effects could be provided. It is not that obvious how to specify custom system triggers. Most of them will be very semantic-specific for a certain DSL. For example the trigger "a state has become active" in a Statechart would require interaction with the simulation engine and hence cannot be implemented only with the knowledge about the certain DSL meta-model and the modeling and visualization framework. Therefore an interface to the "outside" is required, the respective lower level programing environment of the modeling tool.

Such a view management engine could be employed to handle the ideas of semantical focus and context in a general way. It should also allow, via user triggers, to quickly navigate manually through a model, using for example *semantic zooming and panning* where one considers the structure of a model to navigate through it. For example, one would not just change the zoom on a linear percentage scale as is commonly the case, but could also change the zoom by hierarchy level.

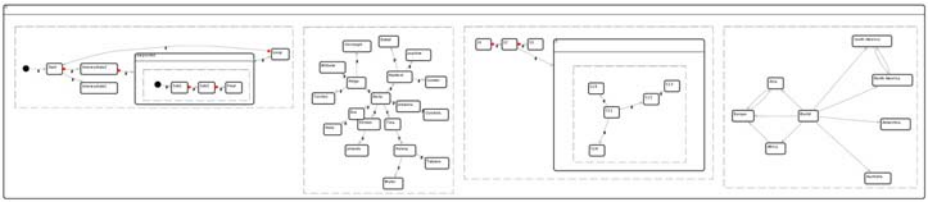


Fig. 9. Meta Layout: Multiple different automatic layout algorithms applied in one diagram, here from left to right GraphViz layered based Sugiyama Layout, the Zest Spring Embedder, a layered layout with radial layouter in child and GraphViz Circo [31]

2.6 Meta Layout

For a given graphical DSL there might be different layouts for the graphical representation conceivable. There may be different automatic layout schemes available, either the same algorithm but with different parametrization options, or completely different layout algorithms. Each layout algorithm results in a different layout style. We denote the process of selecting and combining different already existent layout algorithms as *meta layout*. This should be integrated into the view management.

Note that this is somewhat contradictory with the concept of having a *normal form* [30], where models with the same domain model will have the same graphical representation. The motivation for normalization is to limit ambiguity and subjectiveness when creating or analyzing diagrams. However, it may be hard to find one layout algorithm that provides optimal layout results for all possible applications—even within one DSL. So we may soften the idea of normalization by varying degrees. One could apply different layouters (1) to different models, (2) within one model, in different hierarchy levels and (3) within one model, in different regions of the same hierarchy level (see Fig. 9).

Layouter Choosing Strategies. Having multiple layouters and different regions in the diagram, a question arises: When to apply where what layout? This is answered by *layout choosing strategies*.

The simplest strategy could be to let the user decide. The user manually annotates each part of the model with the specification of which layouter should be used. This way the user would be able to select the best layouters according to his or her personal subjective aesthetic criteria. Additionally, the user could consider application and system specific properties when choosing the layouters.

For a larger benefit, the modeling tool could assist in choosing the right layouter settings by trying to optimize the layout result. The optimization criteria should be provided by the user while the machine should be able to work with them. Possible criteria are *syntactic aesthetic criteria* such as link crossings, link lengths, diagram area, aspect ratio; *semantic aesthetic criteria* such as alignment, symmetry or zoning [32,33]; prescribed *development patterns*; or *model element types*, e. g. graph-based vs. port-based.

3 The Model—Synthesis and Editing

A graphical model is nice to look at, but can be effort-prone to create or change. Common editors have the paradigm of WYSIWYG DND interaction. In general it is desirable to immediately get visualized effects of editing steps in WYSIWYG. However, the way of interaction—DND—is the source of plenty of additional manual editing efforts. Strictly speaking, the term *drag-and-drop* (DND) denotes a specific sequence of steps including the dragging of elements. However, we will refer to DND for all DND *style editing* in current modeling tools. This includes all manual layout positioning of objects on the graphical canvas such as the placement of nodes and edge bendpoints, moving and resizing. Even moving an object by selecting it first and using the arrow keys on the keyboard falls into this category.

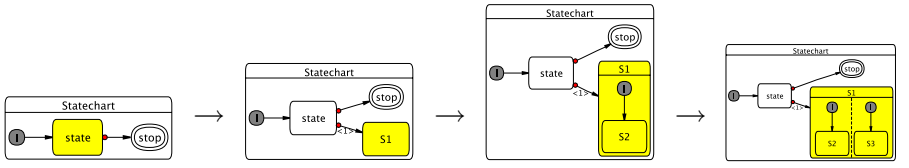
We advocate to try to avoid the tedium induced by DND editing as much as possible, to put back into the focus the system instead of its graphical representation. The basic enabler is the aforementioned capability for automatic layout (Sec. 2.1). One issue here again is the preservation of the mental map of the modeler. In the context of model editing, there exist different schools of thought. One direction argues that the appearance of a model after an edit should be changed only minimally, to preserve the mental map [34]. The other approach is to try to give models a uniform appearance, that "the same should look the same," proposing a *normal form* that is independent of the modeler and the history of the model (see also Sec. 2.6). There the issue of mental map preservation is addressed during the editing step by a morphing animation of the model.

3.1 Structure-Based Editing

The idea of structure-based editing comprises only structural decisions of the developer, which are (1) to select a position in the model topology and (2) to select an operation to apply to the model. This changes only the structure of the model, i. e. its topology, sometimes also referred to as the *domain* or *semantic* model.

The graphical representation also can be updated immediately. The automatic layout has to be applied to create a fresh *view* of the new structure of the model after the user operation. The complexity of the model and the performance of the layout algorithms determine whether it is feasible to apply the layouter after every small editing step in order to get immediate visual feedback. Therefore we eliminate the DND style editing but possibly keep the WYSIWYG nature of the editor. We believe that this immediate visual feedback is valuable enough to put a premium on fast layouting algorithms, even if this might give slightly sub-optimal results.

Structure-Based Editing for Graph-Based Models. For DSLs that are based on graphs we gained some experience from the KIEL project, which applies this paradigm to Statecharts. Graphically they consist of states (nodes),



(a) Simple state is selected (b) New successor state added (c) New state is upgraded (d) After another upgrade

Fig. 10. Example for structure-based editing of a Statechart

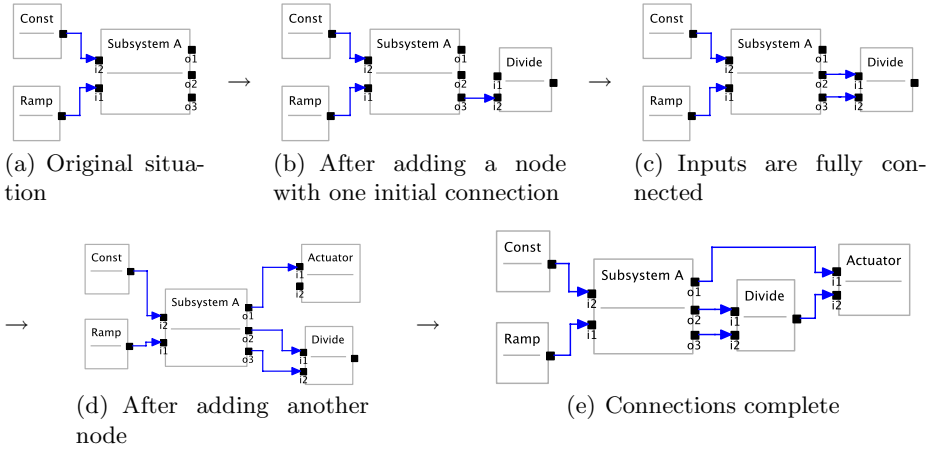
transitions (edges), hierarchy and parallel regions. In this case only a small set of different structural operations are required to create or modify the charts. For a selected state these are only (1) *create a new following state* and (2) *upgrade the state*, as shown in Fig. 10. For transitions the operations are only (1) transition creation and (2) to reverse a transition. Some other "syntactic sugar" can be provided, but nevertheless the operation set is relatively small. Other changes to the model are done afterwards, e. g. changes of labels by filling out form fields.

This paradigm would also apply for other graph-based DSLs because the set of affected model elements in every step is small—up to two. For node operations one node needs to be selected, for edges there are two nodes, source and target.

Structure-Based Editing for Port-Based Models. For *dataflow* models with *ports* (cf. Sec. 2.1) the case is a bit more complex. Especially adding new nodes requires more specification than a simple operation like "add a successor node" can provide. In a graph-based model this operation will generally transform one valid model to another valid model, because it can add a new state and simply connect old and new state with a transition. Port-based models have stricter connection requirements. In general there is an arbitrary set of different kinds of operator nodes; usually this node library is also extensible by the user. Each node has a certain *interface*, i. e. the set of input and output ports that specifies how the node must be connected to other nodes. Hence a new node in the model likely requires not only one but multiple connections which have to be specified not only between the nodes but between specific ports. There are different ways possible for the user interface in this case.

In the first approach the goal is to still provide the diagram itself as the user interface. To support incremental editing, the operation to be performed can be divided in small incremental steps where each does not necessarily lead to a valid dataflow model because it might be not sufficiently connected. After every step the view manager can update the layout and some meaningful graphical representation of the intermediate step is created. An example sequence of such operations is shown in Fig. 11.

In this scenario, the set of operations to connect ports determines the efficiency of creating or editing models. Shortcut operations to connect multiple ports can help to reduce the manual steps. For example the SCADE editor provides the



(a) Original situation

(b) After adding a node with one initial connection

(c) Inputs are fully connected

(d) After adding another node

(e) Connections complete

Fig. 11. Possible structure-based editing steps in a port-based language

operations *connect by rank* and *connect by name* which will interconnect all inputs of one with the outputs of another selected node either by name of the ports or successively by their rank. In SCADE this is not post-processed with the view management, but this can give a first inspiration for the type of connection operations that are helpful.

The operations can be hard-coded for each language or language class. Additionally, the paradigm can be used in conjunction with model transformation frameworks. Especially *in-place transformations* change the underlying domain model by pattern matching where source and target meta-model are the same. Hence the original model is only changed instead of transformed into another DSL. Therefore an in-place transformation framework such as from Taentzer et al. [35] can be used to specify the transformations while the view management with automatic layout adds the graphical feedback to get the full WYSIWYG experience.

3.2 Modification and Deletion

For all possibilities of model changes, the set of model operations must be augmented by operations for removing nodes and connections. Additionally a set of syntactic sugar operations should be provided to manipulate the models efficiently, e. g.

- replace a node by another node of another type,
- replace a connection by a different connection type,
- redirect a connection, or
- insert a new node into one or multiple connections if the port rank fits—i. e. break up the connection into two parts, insert the new node and connect the input and output to the connection endpoints.

This can reduce manual steps especially by keeping attributes of the objects that were manually set after the object creation.

Error Handling. We should learn from best practices in textual programming IDEs and try to adopt features to graphical modeling. For example the *Quick Fix* feature of Eclipse allows beginners to learn textual programming—e. g. Java—in an interactive tutorial-like way. Errors are displayed immediately with the help of incremental continuous compiling. Additionally the UI presents a list of possible solution operations which can be triggered by the user.

Features like this can be incorporated into graphical modeling by orchestration of different building blocks. There are generic modeling frameworks that support model validation such as the Eclipse Modeling Framework (EMF) with its Validation Framework[3]. Hence it is possible to consequently feedback the information about the model consistency to the user. For specific DSLs there should be a set of standard error cases provided together with a set of possible solution operations, again supported by automatic layout of the created solution model.

3.3 Synthesis

With an automatic layout capability, it is not only possible to change models interactively with the developer. One can also synthesize completely new graphical models, including the domain model and its graphical representation. There are multiple scenarios where this *model synthesis* can be of significant benefits and lead to innovative modeling environments.

Textual Modeling. An alternative to the graphical representation of a model still is text. Having information in a textual representation can have many advantages [1,26]. There are already well accepted approaches for *textual modeling* available such as the Textual Concrete Syntax (TCS) [36] or Xtext [37], both frameworks for Eclipse. The developer specifies the meta-model of the DSL and the textual syntax and the framework generates parsers and textual editors. The latter are equipped with convenient features like syntax highlighting, autocompletion, static analysis, structure outline view, source code navigation and folding. Textual models will be parsed into the actual domain model data structures so they can be processed like all other domain models.

The missing link is the one to a graphical model. Here, automatic layout and view management can be used to synthesize the graphical representations from the textual ones. This can be done in different levels of integration:

1. A graphical model is only once *initialized* from the text. Afterwards the graphical model is worked on. Usually there is no way back into the textual model; an exception here is Eclipse.
2. There is a transformation between textual model and graphical model in both directions. This is usually denoted as *round-trip engineering*. Some dedicated commercial tools support this for special DSLs, usually class diagrams, but this is still uncommon.

[3] http://www.eclipse.org/modeling/emf/

3. The tightest integration perfectly synchronizes textual and graphical representation. Hence the user sees two different views and every change in either of the views automatically updates the other view. So working in the views is interchangeable even for small steps. This paradigm has been explored in KIEL for Statecharts and is applicable for other DSLs as well.

To increase the integration further, text and graphics could be mixed in one view. If there is a textual representation for single graphical objects, there could be two different views of the graphical model. One view displays all graphical entities while the other exchanges one of the objects with a text box containing the textual representation of only this model part.

Scalable Models. Model synthesis can be applied together with scripting techniques to create complex and large models according to predefined and parametrizable patterns. Scripts of different flavors could be applied just like *scripts*, *macros* or *templates* in textual languages. This leads to *scalable models*, as investigated in Ptolemy [38]. In this case the scripts that configured the model creation process are in the same graphical syntax as the models themselves. More sophisticated automatic layout techniques could enhance the graphical results. This approach could be applied more generally for arbitrary DSLs and combined with an appropriate user interface.

Pattern-Based Modeling. Development patterns are a common technique in software engineering. When creating behavior diagrams such as Statecharts or dataflow models, one should model common tasks in a common way. This naturally leads to *patterns* for graphical modeling [39,40]. Examples are patterns for error handling, sequencing or loops—depending on the DSLs, many more can be identified. Graphical modeling environments could support the usage of pattern-based development in various ways.

- Design patterns can be highlighted in a model [41].
- A specific pattern can be chosen by the user and parametrized to be added to a graphical model.
- The view management should support user defined automatic layout schemes according to a given pattern. If in a state diagram a loop should be modeled, this could correspond to a pre-defined graphical positioning of the nodes, e. g. in a circle or in a sequence with one back transition.
- Analysis of the model could detect certain patterns for standard operations such as graph transformations [35,42]. Additionally it should be able to layout existing patterns to given pattern layout schemes.

A simple user interface is necessary so even beginners and intermediates can quickly start to employ patterns in their development.

Product Lines. Another use case for proper view management and/or model synthesis are *product lines*. Here, a set of closely related products is offered, where each product likely differs only by some specialization or configuration

from the others. For textual programs, the source code comprises all features, whereas the build process configures different target products with different features deactivated. This could be analogous to the use of pre-processor macros in textual languages, where e. g. an #ifdef macro can hide parts of the program source.

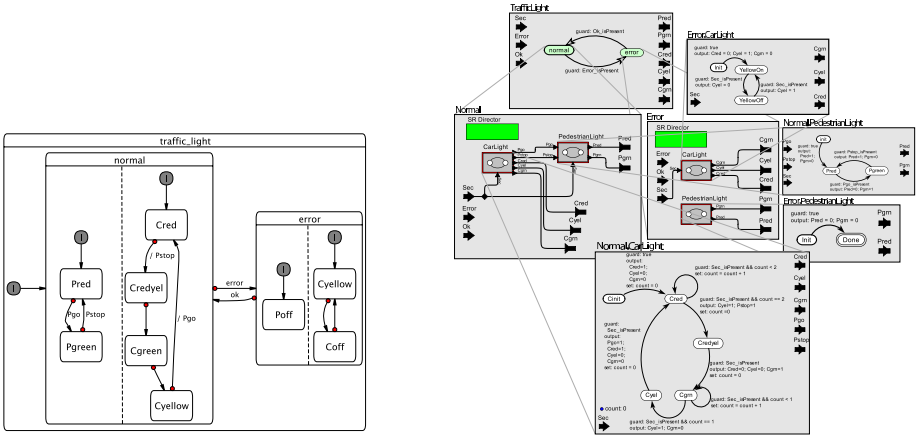
A graphical model can also serve as a *master model* for a product line. To investigate one of the target products and further processing, the final *product model* should be accessible as any other model. To avoid the maintenance of multiple models, the product model should be synthesized from the master model and comprise only the elements necessary for the features of the product. This means omitting certain model objects of the master or configuration of scalable model parts. In certain cases this can be augmented by static analysis to identify the required model parts automatically, e. g., by deactivating superfluous outputs.

3.4 Multi-view Modeling

So far we were considering multiple views only within the same DSL in order to change the levels of detail in certain circumstances to get the best trade-off between overview and details. One can drive the idea of multi-view modeling further by defining completely different views instead of only manipulating the focus and context configuration.

The term *multimodeling* is referred to employing multiple modeling semantics in one single model [43]. For example mixing different semantics such as synchronous data flow with state machines and discrete events or others is a preeminent feature of the Ptolemy modeling framework. This still keeps only one view on the same model, although the model itself is of very heterogeneous character. However, one can for example establish semantical equivalence between Statecharts and mixed synchronous reactive and state machine models [43]. Hence for the same semantics, there exists a Statechart and a Ptolemy model that implements that behavior. This means for the same semantical behavior there exist multiple different graphical representations, each with their advantages and disadvantages. Considering the example in Fig. 12, one might argue that the Statechart model is more compact, but the Ptolemy model makes further information explicit, notably the information flow. We could exploit the equivalence by transforming a Statechart into a Ptolemy model or vice versa—at least for suitable Ptolemy subsets. The disadvantage would be that we still have two completely different models including two different domain models. Both models could be transformed only as whole in a global transformation of all model parts.

An alternative could be to keep only one common domain model and on top of that create two different graphical representations, one for Statecharts and one for Ptolemy. This would be always applicable where one model part can be expressed in multiple ways. Then the model part could have multiple completely different views. The major benefit would be that the different graphical representations could be interchanged in any hierarchy level resulting in a mixed



(a) Statechart model (b) Ptolemy model

Fig. 12. From Statecharts to Ptolemy: both models implement the same behavior [43]

graphical model. The different views could be handled by the view management just as the other views proposed above.

4 The Controller—Interpreting the Model

Sophisticated static analyses can determine properties of a model, for example causality issues for dataflow models [44]. If such an analysis determines certain properties of a set of model elements, it can be used as a trigger for the Meta Layouter in order to get a visual feedback of the analysis. Especially a categorization of model elements in two sets can be interpreted as a categorization into focus and context objects.

4.1 Dual Modeling

The graphical representation depicts the main model objects as nodes, where the containment relations can be reflected by hierarchy in the model. Explicit connections display some other relations between the model objects. However, there is typically a set of model attributes that is hidden in simple property dialogs or simply represented by a label in the graphical representation. Relations between those attributes are usually not visible.

We propose a dynamic extension of the graphical representation by its *dual model*, i. e. a graphical representation of the relations between referenced objects where this reference is not yet visualized. We again examine the example of Statecharts. The dual model of a Statechart is a graph where the transition labels are the nodes and the relations between guards and actions form the connections. The graph shows which transition produce triggers and which ones

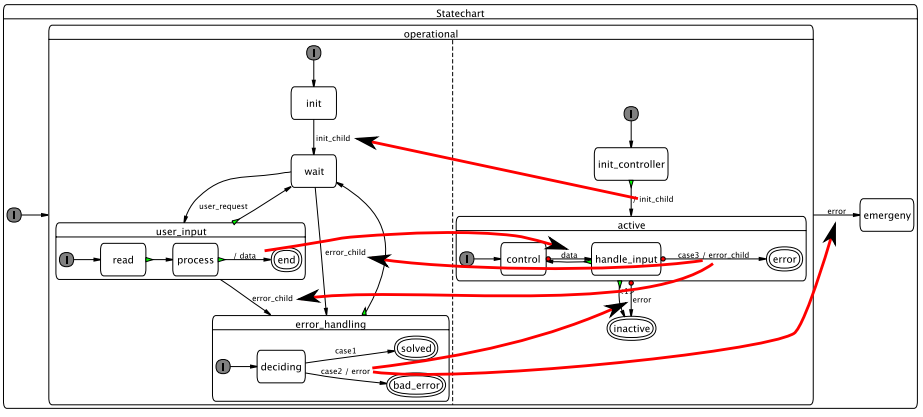


Fig. 13. Dual Model for Statecharts: Two parallel controllers communicate via broadcast. The dataflow is displayed as an overlay of the original control flow graphical representation.

read those triggers. It makes explicit how the broadcast communication is used by showing the flow of data and signals in the model. By graphically overlaying the original graphical representation with its dual model, we reuse the same graphical view in order to keep the mind map within the user, as illustrated in Fig. 13.

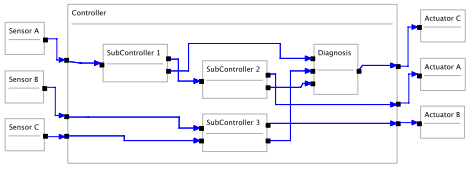
The *dual model* methodology should not only be helpful for Statecharts, but applies to very different types models. References to other model parts are quite common where an explicit graphical representation is omitted for the sake of clarity in the original model.

4.2 Dynamic Behavior Analysis

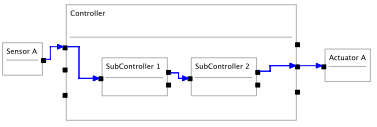
We usually distinguish the *structure* and the *behavior* of a model. To validate behavior, it is common practice to employ simulations prior to physical deployment. Therefore we employ DSLs with known specified semantics such that the models can be executed.

Simulation Management. Employing the meta layouter during testing gives us the same benefits as for simple manual browsing, as interesting parts can be put into the focus while the context is still visible. Additionally, a simulation run gains a new dimension: time. Hence there might be times where nothing of relevance happens and other points in time with interesting events. The problem is to determine "interesting" parts and times during simulation.

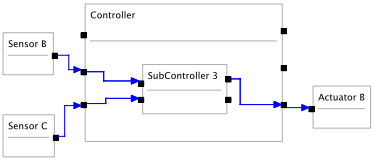
Therefore we propose to extend the meta layout view management by *simulation management*. It defines an additional set of system events for triggering view management effects and additional effects for manipulating simulation time.



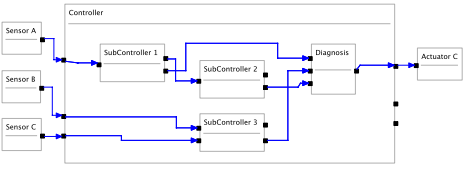
(a) Original controller



(b) All dependencies of Actuator A, two sub-controllers and only Sensor A



(c) Dependencies of Actuator B



(d) Dependencies of Actuator C, which are all prior inputs

Fig. 14. View Management in a dataflow language for some embedded controller with three sensors and three actuators

Both simulation triggers and effects are highly dependent on the language semantics. Hence a simulation manager is usually only applicable for a small set of DSLs.

Visual Breakpoints. Simulation *triggers* are customizable conditions over internal states and variables of the simulation. Hence both the specification and the interpretation of those triggers require access to the semantics of the model and the simulation engine. The triggers cause effects, on the one hand usual view changing effects, such as graphical focus change events, on the other hand *simulation effects* that alter the behavior of simulation time, such as simulation pause or stop.

A simulation manager should allow to specify *visual breakpoints*, the combination of a specific target view with the condition under which this view will be shown and possibly the suspension of the simulation to give time for analysis of the situation. A properly configured simulation manager knows what "interesting" items are, both in time and model objects. So during simulation a user always gets to see the right parts of interest without any manual user interaction; no manual navigation actions are required.

An example for dataflow diagrams is shown in Fig. 14. Here a focus is set to one actuator and all components in the dataflow towards that actuator. Other components are filtered. This results in tidy diagrams that illustrate specific aspects, e. g. for analyzing Actuator B. During a simulation run, the respective view could be shown, whenever some specific value is received by one of the actuators. The way of actually displaying the data is another issue but could be integrated into the diagram. The dynamic focus and context technique implemented in KIEL for Statecharts (cf. Sec. 2.4) could be implemented in a straight

forward fashion by adding simulation events for every state change and setting the set of focus objects to the the active states.

Simulation Tracking and Control. It is common practice to show (highlight) the current *state* of a system. In some areas, it is also common to show the current *change of state* (e. g., a transition in a Statechart). There are natural extensions that one could consider, such as showing the *recent past* (e. g., the last n states), or the *possible future* (states that might be reachable in the next n steps, this would require some kind of static/dynamic analysis).

A desirable feature is to be able to not just run a simulation and to stop it at certain points, but also to step backwards again. This *tape recorder paradigm* has already been integrated into some modeling tools, e. g., Statemate [25].

5 Conclusions

We have presented an overview of different aspects of modeling pragmatics. A guiding principle has been the model view controller paradigm, which has been quite successful in software engineering and which we believe has much to offer in the world of model based design as well.

We consider automatic layout of the graphical representation to be one of the basic key enablers for good pragmatics. We build upon layouters by dynamic filters that reduce the complexity of diagrams and focus and context as a special case of such filters. A view management engine organizes different dynamic views synthesized with filters in order to assist the user in seeing the "interesting" parts of the model. We extend the view management by meta layout, which plays with different layout styles even within different parts of one graphical model in order to get optimal layout results.

With these building-blocks we support a set of use-cases in the modeling process that will help us to cope with very large model instances. For creation and modification we propose structure-based editing to free the user of many manual effort prone tasks. Auto-layout enables graphical model synthesis and opens the door for perfectly synchronized textual and graphical representations, scalable models, pattern-based modeling and support for product lines.

This survey cannot hope to be complete in any way. What we do hope to achieve is to raise the level of awareness about the importance and possibilities of modeling pragmatics in general. In a way, this paper might thus be regarded as a (partial) road map for possible future developments in modeling pragmatics.

References

1. Gurr, C.A.: Effective diagrammatic communication: Syntactic, semantic and pragmatic issues. Journal of Visual Languages and Computing 10(4), 317–342 (1999)
2. Petre, M.: Why looking isn't always seeing: Readership skills and graphical programming. Communications of the ACM 38(6), 33–44 (1995)

3. Prochnow, S., von Hanxleden, R.: Statechart development beyond WYSIWYG. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 635–649. Springer, Heidelberg (2007)
4. Reenskaug, T.: Models – Views – Controllers. Technical report, Xerox PARC technical note (December 1979)
5. Kopetz, H.: The complexity challenge in embedded system design. Research Report 55/2007, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria (2007)
6. Estefan, J.: Survey of model-based systems engineering (MBSE) methodologies, Rev. B. Technical report, INCOSE MBSE Focus Group (May 2008)
7. Carloni, L., Passerone, R., Pinto, A., Sangiovanni-Vincentelli, A.: Languages and tools for hybrid systems design. Foundations and Trends in Design Automation 1(1), 1–204 (2006)
8. Prochnow, S., von Hanxleden, R.: Visualisierung komplexer reaktiver Systeme – Annotierte Bibliographie. Technical Report 0406, Christian-Albrechts-Universität Kiel, Department of Computer Science, Kiel, Germany (June 2004)
9. Di Battista, G., Eades, P., Tamassia, R., Tollis, I.G.: Algorithms for drawing graphs: An annotated bibliography. Computational Geometry: Theory and Applications 4, 235–282 (1994)
10. Battista, G.D., Eades, P., Tamassia, R., Tollis, I.G.: Graph Drawing: Algorithms for the Visualization of Graphs. Prentice Hall, Englewood Cliffs (1999)
11. Coleman, M.K., Parker, D.S.: Aesthetics-based graph layout for human consumption. Software – Practice and Experience 26(12), 1415–1438 (1996)
12. Ware, C., Purchase, H., Colpoys, L., McGill, M.: Cognitive measurements of graph aesthetics. Information Visualization 1(2), 103–110 (2002)
13. Völcker, J.: A quantitative analysis of Statechart aesthetics and Statechart development methods. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science (May 2008), http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/jovo-dt.pdf
14. Green, T.R.G., Petre, M.: Usability analysis of visual programming environments: A 'cognitive dimensions' framework. J. Visual Languages and Computing 7(2), 131–174 (1996)
15. Gutwenger, C., Jünger, M., Klein, K., Kupke, J., Leipert, S., Mutzel, P.: A new approach for visualizing UML class diagrams. In: SoftVis 2003: Proceedings of the 2003 ACM Symposium on Software Visualization, pp. 179–188. ACM, New York (2003)
16. Lee, E.A., Neuendorffer, S., Wirthlin, M.J.: Actor-oriented design of embedded hardware and software systems. Journal of Circuits, Systems, and Computers 12, 231–260 (2003)
17. Eiglsperger, M., Fößmeier, U., Kaufmann, M.: Orthogonal graph drawing with constraints. In: SODA 2000: Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 3–11. SIAM, Philadelphia (2000)
18. Card, S.K., Mackinlay, J., Shneiderman, B.: Readings in Information Visualization: Using Vision to Think. Morgan Kaufmann, San Francisco (1999)
19. Tiedje, M., Traulsen, C.: Designing a reactive processor with Esterel v7. In: Proceedings of the Workshop on Model-Driven High-Level Programming of Embedded Systems (SLA++P 2008), Budapest, Hungary (April 2008)

20. Leung, Y., Wilson, G.: WinFold: a folding editor for collaborative writing Communications. In: APCC/OECC 1999. Fifth Asia-Pacific Conference on Communications and Fourth Optoelectronics and Communications Conference, vol. 2, pp. 1073–1078 (1999)
21. Dogrusöz, U., Kakoulis, K.G., Madden, B., Tollis, I.G.: On labeling in graph visualization. Inf. Sci. 177(12), 2459–2472 (2007)
22. van Dijk, S., van Kreveld, M., Strijk, T., Wolff, A.: Towards an evaluation of quality for names placement methods. International Journal of Geographical Information Systems (2002)
23. Kakoulis, K.G., Tollis, I.G.: On the edge label placement problem. In: North, S.C. (ed.) GD 1996. LNCS, vol. 1190, pp. 241–256. Springer, Heidelberg (1997)
24. Wong, P.C., Mackey, P., Perrine, K., Eagan, J., Foote, H., Thomas, J.: Dynamic visualization of graphs with extended labels. In: INFOVIS 2005: Proceedings of the 2005 IEEE Symposium on Information Visualization, Washington, DC, USA, p. 10. IEEE Computer Society, Los Alamitos (2005)
25. Harel, D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R., Shtull-Trauring, A., Trakhtenbrot, M.: Statemate: A working environment for the development of complex reactive systems. IEEE Transactions on Software Engineering 16(4), 403–414 (1990)
26. Prochnow, S., Traulsen, C., von Hanxleden, R.: Synthesizing Safe State Machines from Esterel. In: Proceedings of ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2006), Ottawa, Canada (June 2006)
27. Köth, O., Minas, M.: Structure, Abstraction, and Direct Manipulation in Diagram Editors. In: Hegarty, M., Meyer, B., Narayanan, N.H. (eds.) Diagrams 2002. LNCS (LNAI), vol. 2317, pp. 290–304. Springer, Heidelberg (2002)
28. Branke, J.: Dynamic graph drawing. In: Kaufmann, M., Wagner, D. (eds.) Drawing Graphs: Methods and Models. LNCS, vol. 2025, p. 228. Springer, Heidelberg (2001)
29. Musial, B., Jacobs, T.: Application of focus + context to UML. In: APVis 2003: Proceedings of the Asia-Pacific symposium on Information visualisation, Darlinghurst, Australia, pp. 75–80. Australian Computer Society, Inc. (2003)
30. Prochnow, S., von Hanxleden, R.: Comfortable modeling of complex reactive systems. In: Proceedings of Design, Automation and Test in Europe (DATE 2006), Munich, Germany (March 2006)
31. Schipper, A.: Layout and Visual Comparison of Statecharts. Diploma thesis, Christian-Albrechts-Universität zu Kiel (December 2008), http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/ars-dt.pdf
32. Kosak, C., Marks, J., Shieber, S.: Automating the layout of network diagrams with specified visual organization. Transactions on Systems, Man and Cybernetics 24(3), 440–454 (1994)
33. Purchase, H.C.: Which aesthetic has the greatest effect on human understanding? In: Di Battista, G. (ed.) GD 1997. LNCS, vol. 1353, pp. 248–261. Springer, Heidelberg (1997)
34. Castelló, R., Mili, R., Tollis, I.G.: A framework for the static and interactive visualization for statecharts. Journal of Graph Algorithms and Applications 6(3), 313–351 (2002)
35. Biermann, E., Ehrig, K., Khler, C., Kuhns, G., Taentzer, G., Weiss, E.: Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 425–439. Springer, Heidelberg (2006)

36. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In: GPCE 2006: Proceedings of the 5th International Conference on Generative Programming and Component Engineering, pp. 249–254. ACM, New York (2006)
37. Efftinge, S., Voelter, M.: oAW xText: A framework for textual DSLs. In: Eclipse Summit Europe, Esslingen, Germany (October 2006)
38. Feng, T.H., Lee, E.A.: Scalable models using model transformation. In: 1st International Workshop on Model Based Architecting and Construction of Embedded Systems (ACES^{M}B) (September 2008)
39. Gamma, E., Helm, R., Johnson, R., Vlissides, J.M.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)
40. Douglass, B.P.: Real-time Design Patterns: Robust Scalable Architecture for Real-time Systems. Addison-Wesley, Reading (2003)
41. Peters, A.K.: Musterbasiertes Layout von Statecharts. Masters thesis, Universität Hamburg, Falkultät für Mathematik, Informatik und Naturwissenschaften, Department Informatik (June 2008)
42. Karsai, G., Agrawal, A., Shi, F., Sprinkle, J.: On the use of graph transformation in the formal specification of model interpreters. Journal of Universal Computer Science 9(11), 1296–1321 (2003)
43. Brooks, C., Cheng, C.H.P., Feng, T.H., Lee, E.A., von Hanxleden, R.: Model engineering using multimodeling. In: Proceedings of the 1st International Workshop on Model Co-Evolution and Consistency Management (MCCM 2008), a workshop at MODELS 2008, Toulouse (September 2008)
44. Zhou, Y., Lee, E.A.: Causality interfaces for actor networks. ACM Transactions on Embedded Computing Systems (TECS), 1–35 (April 2008)

Modelling and Verification of Relay Interlocking Systems

Anne E. Haxthausen, Marie Le Bliguet, and Andreas A. Kjær

DTU Informatics
Department of Informatics and Mathematical Modelling
Technical University of Denmark, DK-2800 Lyngby, Denmark
ah@imm.dtu.dk

Abstract. This paper describes how relay interlocking systems as used by the Danish railways can be formally modelled and verified. Such systems are documented by circuit diagrams describing their static layout. It is explained how to derive a state transition system model for the dynamic behaviour of a relay system from such diagrams. Safety properties are identified and formalised as LTL formulae. Model checking is finally used to verify that a model satisfies the safety properties. The method is tested for an existing station in Denmark.

1 Introduction

The task of railway interlocking systems is to control signals and points in such a way that the railway traffic keeps safe, i.e. collisions and derailing of trains are avoided.

The Danish railways use a route based approach to interlocking. The idea is that two trains must never drive on two conflicting (i.e. overlapping) train routes at the same time (to avoid collisions), and a train must only drive on a train route in which the points are locked in the positions that allow travel along the route (to avoid derailing).

For many Danish railway stations the interlocking systems are implemented using electrical circuits containing relays. The Danish railways document their relay systems by drawings (diagrams of the electrical circuits) and currently the only way to verify desired properties of the relay systems is to inspect the drawings and manually draw conclusions. This is hard to do as the number of drawings for a single system is very high and the logic described in each of them is complicated with many mutual dependencies. Certainly such a manual verification process is not only difficult and time consuming, but may also be error prone. This is not satisfactory for a safety-critical system.

In this paper we will report on the first results from a project the goal of which is to automate the verification process. Our approach to verify a relay interlocking system has been as follows: first a formal state transition system model of the behaviour of a relay interlocking system is specified in RSL-SAL[1],

[1] RSL-SAL [8,9] is an extension of the RAISE Specification Language [10] with constructs for defining state transition systems and desired properties of these.

C. Choppy and O. Sokolsky (Eds.): Monterey Workshop 2008, LNCS 6028, pp. 141–153, 2010.

and the safety properties are formalised as LTL formulae, also in RSL-SAL. Then the RSL-SAL model and properties are translated into SAL[1,4] using the RAISE tools, and the verification that the model satisfies the properties is performed using a model checker for SAL.

Formal modelling and verification have previously been performed for other kinds of Danish interlocking systems (electronic ones), see [7,6], however, the work reported in this paper is, to the authors' knowledge, the first formal model and verification made for Danish *relay* interlocking systems. For other complementary and competing approaches for the development of railway control systems the reader is referred to the contributions in [13,11,12,5], and for a survey of results and current trends the reader is referred to the paper [2].

First, in Section 2, we give an introduction to train route based interlocking. Then in Sections 3–4, we describe the syntax of relay circuit diagrams and explain how a behavioural model of a relay system can be derived from its diagrams. In Section 5 we sketch how to model behaviour of the environment that interacts with the relay interlocking system. In Sections 6–7 we give examples of how to formalise safety related requirements and other desired properties. Then, in Section 8, we report on how we have applied the presented method to create a model and requirements for Stenstrup station in Denmark, and how we have used the SAL model checker to check that the model satisfies the requirements. Finally, in Section 9 some conclusions are drawn.

2 Train Route Based Interlocking

In this section we introduce the concepts of train route based interlocking systems.

2.1 Equipment at a Station

Train route based interlocking systems use various track-side equipment to monitor and control trains:

Track circuits: The railway tracks are divided into sections each having equipment (a circuit) for train detection. The interlocking system uses this for monitoring the occupancy status of the individual track sections.

Points: Tracks are joined at points which can guide trains into different directions depending on the position of the point. An operator can switch the points by pushing some **buttons**. The interlocking system monitors and controls the positions of points.

Signals: Signals are placed at the entrance of some track sections. They can show GO and STOP aspects. The interlocking system sets the signals to inform the train drivers whether they are allowed to enter these sections.

2.2 Train Routes and Their Use

The stations we are considering in this paper use a *route based* approach to interlocking. The basic ideas of this approach are:

- Trains should drive on *routes* through the network.
- Each route is covered by an entrance signal that informs whether it is allowed for a train to enter the route or not. The trains must respect the signals.
- Two trains must never be allowed to drive on conflicting (i.e. overlapping) routes at the same time. *(To prevent collisions.)*
- Before a train is allowed to enter a route, the points must be locked in positions making the route connected (i.e. it is physically possible to go from one end of the route to the other end without derailing), and the route must be empty (i.e. there are no trains on the route). *(To prevent derailing and collisions, respectively.)*
- The points of a route must not be switched while a train is driving on the route. *(To prevent derailing.)*

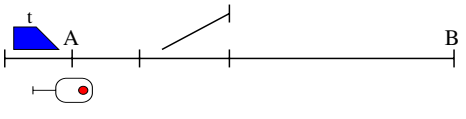
For each station to be controlled by an interlocking system, an *interlocking table* is used to specify routes through that station. Such tables define for each train route

- which positions points must have for the route to be *connected*,
- which track sections must be unoccupied for the route to be *empty*, and
- which settings of signals are required for the route to be *open* (i.e. for allowing trains to enter the route).

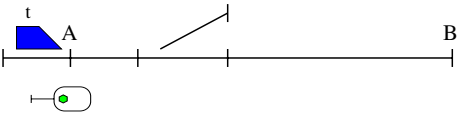
The tables also define which train routes are *conflicting*.

Sample Scenario. It is outside the scope of this paper to give all the details of the interlocking protocol, but in this section we will show a typical scenario for a train t to go along a route R from A to B. The route consists of three track sections of which the middle one is a point. In all figures below, the point will be shown in its straight position. A signal is placed at the route entrance A. When the rightmost lamp is red it means STOP, and when the leftmost lamp is green it means GO.

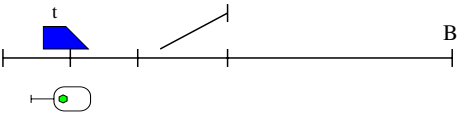
- Initially the train waits in front of the entrance signal that is set to STOP:



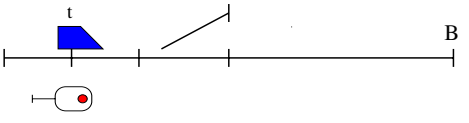
- As a first action the operator sets the points to the positions required for the route R to be connected. In this example, there is only one point involved in the route, and it is set to its straight position.
- Then the operator pushes a button to lock R.
- When the system detects that the button is pushed, it locks the route (after having checked that no conflicting routes are locked and that the points are in the positions that ensure R is connected).
- Then the system sets the entrance signal to GO (after having checked that R is empty):



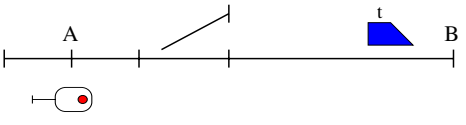
– Now the train enters the route:



– When the system detects that the first track section of the route is occupied, it sets the entrance signal to STOP (to prevent additional trains to enter the route):



– The system unlocks the route when the train is entirely on the last section of R:



Now if a second train is to use the route, the protocol is to be followed again. It would be possible to lock the route again, but the signal can first be set to GO when the first train has completely left the route (i.e. is on the right-hand side of B).

3 Relay Circuits and Diagrams

3.1 Relay Circuits

The interlocking systems we are considering are implemented by electrical circuits. The circuits are made up of components such as power supplies, relays, contacts, and buttons, connected by wires.

A *relay* is an electrical switch operated by an electromagnet to connect or disconnect a number of contacts in a circuit. When current goes through the relay, the magnet is *drawn* and some of the associated contacts are connected (these contacts are said to be *upper contacts*) while others (the *lower contacts*) are disconnected. When no current goes through the relay, the magnet is *dropped* and the associated upper and lower contacts will be disconnected and connected,

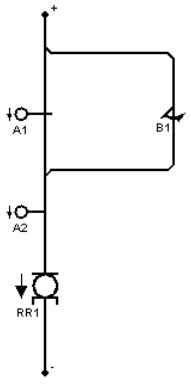


Fig. 1. Diagram for circuit controlling relay $RR1$

respectively. When contacts are connected/disconnected this may imply that sub-circuits containing these contacts become live/dead. This again may imply that relays of these sub-circuits are drawn or dropped.

The system can get input from the environment:

- buttons can be pushed (and later released) by an operator
- track relays are dropped/drawn when trains enter/leave track sections
- point relays are dropped/drawn when points are moved into a new position

The track relays and point relays are said to be *external*, while the relays controlled by the interlocking system are said to be *internal*.

An input may lead to a chain of internal events: relays that are drawn and dropped. In practise such chains are very short and take almost no time. Therefore, in this work we will assume that new inputs do not happen until no more internal events can happen.

3.2 Diagrams

The Danish railways use diagrams to document the electrical circuits of a relay system.

For each internal relay one of the diagrams shows the sub-circuit that controls that relay. An example of such a diagram is shown in Figure 1. This diagram shows the sub-circuit controlling a relay named $RR1$. The circuit consists of a number of components connected by wires. The wires are depicted as black lines. At the top is the positive pole and at the bottom is the negative pole of the power supply. Relay $RR1$ is shown using this signature:

The downwards arrow informs that in the initial state this relay is dropped. (If it had been drawn the arrow would have been upwards.) A number of contacts belonging to other relays occur in this circuit. E.g. a contact belonging to a relay named $A1$ is shown using this signature:

The downwards arrow informs that in the initial state relay $A1$ is dropped. The horizontal bar breaks the wire – this indicates that the contact is disconnected in the initial state. If it had not been breaking the wire it would have indicated that the contact had been connected in the initial state. Also a button $B1$ is shown on the diagram using this signature:

A pushed button is shown by this signature:

3.3 Electrical Behaviour

In this section we present an example of the dynamics of a circuit.

The example shows a scenario where a button of a circuit is pushed. In Figure 2 the first four states of the circuit in this scenario are visualised on a diagram of the circuit. Wires that are current carrying are shown by a grey colour. State 0 is the initial state. In the initial state, no wires are current carrying. When the button is pushed, current is going from plus to minus through relay 37, see state 1. As a consequence of this, relay 37 is drawn and its associated upper contact becomes connected, opening a second path of current from plus to minus through relay 33, see state 2. As current is going through relay 33, this will be drawn, see state 3. In state 3 no more internal events can happen.

4 Modelling a Relay Interlocking System

We are now going to explain how one from circuit diagrams can derive a state transition system model of a relay system.

State Space. The following components may change state:

- buttons may be pushed or released,
- relays may be drawn or dropped,
- contacts may be connected or disconnected.

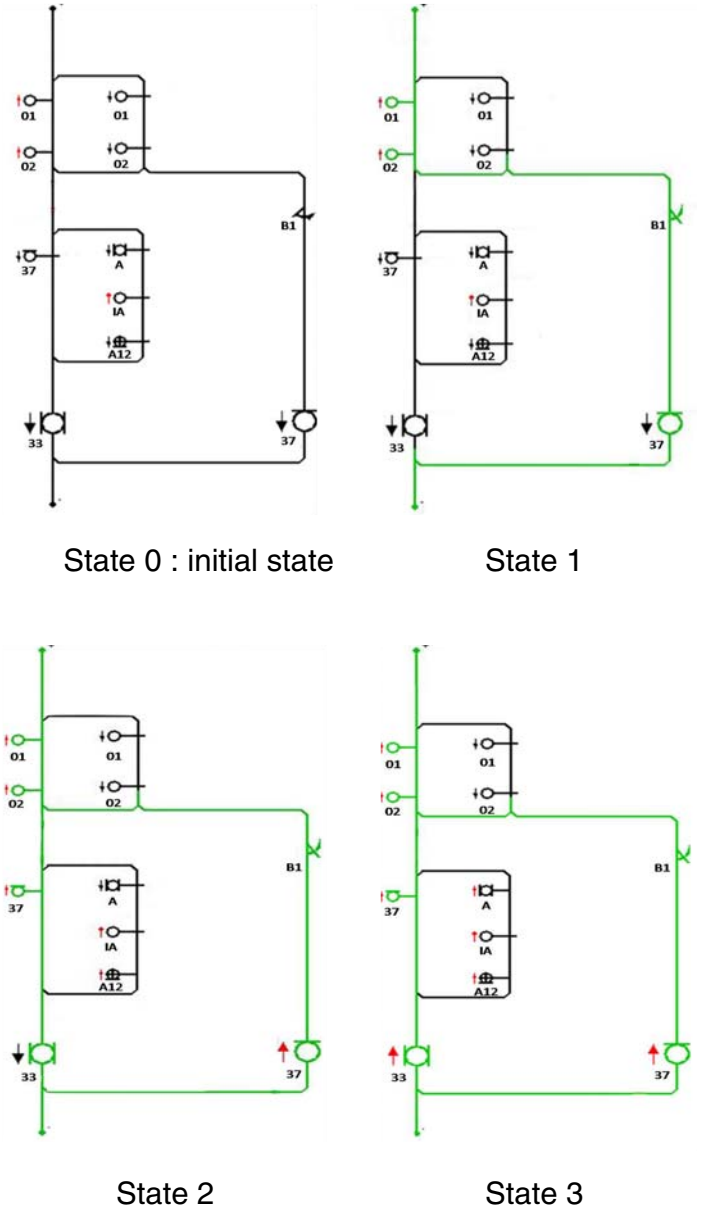


Fig. 2. A state sequence for a circuit

We introduce a Boolean variable for each button and for each relay. When a variable is **true** it means pushed and drawn, respectively, and **false** means released and dropped, respectively. We do not need to introduce variables for contacts as their state can be derived from the other states. The initial state of the buttons is **false**, and the initial states of the internal relays is as stated on the diagrams.

Transition Rules for Internal Relays. For each internal relay r there are two rules, one for drawing it and one for dropping it:

$$[\,\text{draw\_r}\,]\ {\sim}\text{r} \wedge\ \ \text{conducting\_r} \rightarrow \text{r}' = \textbf{true},$$
$$[\,\text{drop\_r}\,]\ \text{r} \wedge {\sim}\,\text{conducting\_r} \rightarrow \text{r}' = \textbf{false}$$

The first rule expresses that r can be set to true (meaning that r becomes drawn) when r is dropped and conducting current, while the second rule expresses that r can be set to false (meaning that r becomes dropped) when r is drawn and not conducting current.

The condition *conducting_r* for current to go through a relay r is a logical formula determined as follows.

Current will go through the relay if there is a path from the positive pole to the negative pole that goes through the relay, and all contacts within this path are connected and all buttons are pushed.

Now for a given relay there are several potential paths, $p_1, ..., p_n$, for current to go through it. For each potential path p_i we express the condition *cp_i* for that path to be conductive. Then the condition for the relay to be conducting is the disjunction of these conditions:

$$\text{conducting\_r} = \text{cp\_1} \vee ... \vee \text{cp\_n}$$

The condition for a potential path to be conductive is a conjunction of conditions for its contacts to be connected and its buttons to be pushed. The condition for a button b to be pushed is b. The condition for an upper contact and a lower contact belonging to relay r to be connected is r and $\neg r$, respectively.

As an example, for relay $RR1$, we can from the diagram in Figure 1 deduce the following condition for current to go through the relay:

$$\text{conducting\_RR1} = (\text{A1} \wedge {\sim}\text{A2}) \vee (\text{B1} \wedge {\sim}\text{A2})$$

5 Modelling the Environment

The system is said to be in an *idle* state, when it is ready for input, i.e. no internal event is possible, cf. the assumption mentioned in the end of Section 3.1. In order to easily keep track of when the system is idle, we add to the state space a Boolean variable *idle*.

The rule for the system to become idle is of the form

$${\sim}\text{idle} \wedge {\sim}\text{c} \rightarrow \text{idle}' = \textbf{true}$$

where c is the disjunction of all guards for internal relay events.

Transition rules for external events will then always take the form:

$$\text{idle} \wedge ... \rightarrow ..., \text{idle}' = \textbf{false}$$

Below we consider transition rules for track sections and points.

Transition Rules for Track Sections. The tracks are divided into track sections, each of which is monitored by an associated track relay. The track relays are controlled by the movement of trains: when a train enters or leaves a track section, the associated track relay is dropped and drawn, respectively. Therefore the rules for track relays should reflect possible train movements and will depend on the track layout for the station and in particular on the placement of signals. The rules reflect that trains

- only enter the station from entry sections
- only leave the station at exit sections
- follow the tracks
- do not pass signals showing STOP
- do not change direction while using a route
- do not split

Transition Rules for Points. Point control is also done using relay circuits. This means that in principle we could have used our techniques for deriving transition rules from relay circuit diagrams that describe the behaviour of points. However, as these relay circuits are rather complex, we have in this work decided to directly formulate transition rules that express the desired point behaviour without inspecting the diagrams. It is then another task to verify that the circuits correctly implement the point control expressed by our rules. As the points we are considering all use a standard implementation that has been validated and used for more than 60 years, we have good confidence that they behave correctly.

Points can be in one of three positions: plus (straight), minus (branching), and intermediate (between plus and minus). Each point p has its position monitored by two relays, $p+$ and $p-$. p is in its plus position when $p+$ is drawn (and $p-$ is dropped), p is in its minus position when $p-$ is drawn (and $p+$ is dropped), and p is in its intermediate position when $p+$ and $p-$ are both dropped. The point also constitutes a track section, so as for all other track sections there is an associated relay tp that is dropped when p is occupied by a train.

A point is only allowed to be switched, when (1) the track section tp of the point is unoccupied, and (2) all routes that include the point are unlocked (this condition can be determined as for each route there is a relay that is drawn when the route is unlocked).

There are four rules associated with a point p. Here are the rules for switching p from its minus to its intermediate position, and for switching p from its intermediate to its minus position:

[p_minus_to_intermediate]
idle $\wedge$ p$-$ $\wedge$ tp $\wedge$ r1 $\wedge$ ... $\wedge$ rn $\rightarrow$ p$-'$ = **false**, idle$'$ = **false**,

[p_intermediate_to_minus]
idle $\wedge$ $\sim$p$-$ $\wedge$ $\sim$p+ $\wedge$ tp $\wedge$ r1 $\wedge$ ... $\wedge$ rn $\rightarrow$ p$-'$ = **true**, idle$'$ = **false**

where $r1, ..., rn$ are the "locking relays" for those routes that use the point.

There are two similar rules for switching between the plus position and intermediate position.

6 Confidence Conditions

In this section we will consider some properties that are desired for relay circuits.

6.1 No Internal Cycles

A desirable property of a relay circuit is the absence of internal cycles where the same sequence of internal relay events is repeated over and over again as the reaction to an input to the system. In our context, the absence of internal cycles is equivalent to require that the system will always eventually become idle again. In LTL, this can be formulated as follows:

G(F(idle))

6.2 No Critical Races

It is also desirable that the system is *deterministic*, i.e. whenever the system is given the same input in the same state, the next idle state is the same. There is the danger that this is not the case, if there is a (reachable) state for which several internal relay events are possible. For such a state the next idle state might depend on which of the possible relay events happens first. Such a case is also called a *race condition*. In the following we will look for an LTL formula that expresses that there are no race conditions.

Consider two internal relay events described by rules of the form

g1 $\rightarrow$ r1$'$ = e1,
g2 $\rightarrow$ r2$'$ = e2

and assume that the system is in a state where both events are possible (because both guards $g1$ and $g2$ are true). The two events must be associated with two different relays $r1$ and $r2$ as the guards for drawing and for dropping the same relay r can never be true at the same time (as they require r to be false and true, respectively), and therefore the two events change different variables $r1$ and $r2$. Hence, to avoid a race condition, it is sufficient to require that if one event happens first, it must still be possible for the other event to happen. (In this way both events will happen before the next idle state, and the order does

not matter, as they change different variables.) In other terms taking one of the transitions must not falsify the guard of the other transition. This leads to the idea of having the following confidence condition for each relay r:

$$\mathrm{G}(\mathrm{can\_draw(r)} \Rightarrow \mathrm{X}(\sim \mathrm{can\_draw(r)} \Rightarrow \mathrm{r}))$$

where $can\_draw(r)$ is the guard for drawing the relay r. The condition expresses that for any reachable state in which r can be drawn, the following will hold for the next state: if r can't anymore be drawn then its is because it has just been drawn. In other words, only the drawing event for r can falsify the $can\_draw(r)$ guard. We have similar confidence conditions for dropping relays.

7 Safety Related Requirements

In this section we give examples of how to formalise safety related requirements that an interlocking system must satisfy. We distinguish between requirements at two different levels of abstraction. At the highest level of abstraction we have the ultimate, functional requirements that are independent of the chosen interlocking approach, while at the lower level we have requirements specific for the route based interlocking approach. The requirements are formalised as LTL formulae in RSL-SAL.

7.1 Higher Level Requirements

The requirements at the higher level are concerned with the avoidance of derailing and collisions.

As an example, the following condition expresses the higher level requirement that when a train is occupying a point p, it must not be in a switching state:

$$[\,\mathrm{no\_derailing\_p}\,]\ \mathrm{G}(\sim \mathrm{tp} \Rightarrow (\mathrm{p}+ \vee \mathrm{p}-))$$

Here tp is the track relay of p, and $p+$ and $p-$ are the two (external) point relays associated with p.

7.2 Lower Level Requirements

The requirements at the lower level are concerned with conditions specific for train route based interlocking: they express that the route based protocol has to be followed.

Examples of lower requirements are:

1. Two conflicting train routes must not be locked at the same time.
2. When a route is locked, the points must be in positions making the route connected.
3. When a route is locked, it must retain being locked until the last section of the route is occupied.

4. When a route becomes open (i.e. the entrance signal is set to GO), the route must be locked and empty. This implies that when a train enters a route, the signal must change to STOP.

For a specific station, these requirements can be formalised as LTL formulae by using the data of the interlocking table for that station. For instance, for each pair of routes $r1$ and $r2$ that are conflicting according to the interlocking table, the first requirement gives rise to the following condition:

$$[\,\text{no\_r1\_r2\_conflict}\,]\ \mathrm{G}(\mathrm{r1} \vee \mathrm{r2})$$

where $r1$ and $r2$ are relays that are drawn when routes $r1$ and $r2$ are unlocked, respectively.

8 Experiments

Following the principles explained in previous sections, we made a state transition system model of the interlocking system for Stenstrup station and formulated confidence conditions and safety conditions. The station layout for Stenstrup is shown in Figure 3. The system had 4 buttons, 46 internal relays and 10 external relays. The transition system model contained > 61 Boolean variables, 92 transition rules for internal relays and additional rules for the environment. There were 100 confidence conditions and 36 safety conditions. We translated the model and conditions into SAL using the RAISE tools, and then we used the SAL model checker to verify that the model satisfies all the conditions.

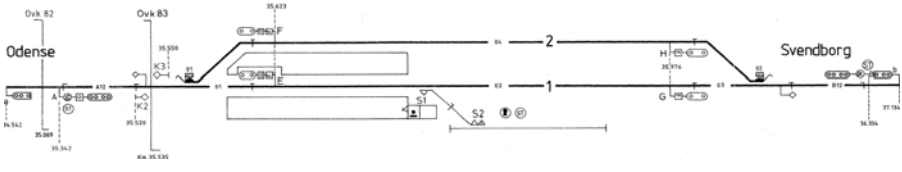


Fig. 3. Stenstrup station

9 Conclusions

In this paper we have explained how relay interlocking systems as used by the Danish railways can be formally verified by model checking. The method has successfully been applied to the relay interlocking system for Stenstrup station in Denmark. Further information on this work can be found in [3].

It should be mentioned that we have also implemented a tool (see [3]) that generates models and confidence conditions from relay circuit diagrams using the general principles described in this paper. We plan in the future also to make a tool that can extract safety conditions from interlocking tables.

Acknowledgements. The authors would like to thank Kirsten Mark Hansen, Banedanmark, for providing the initial idea for this project and for many valuable discussions and suggestions. Our thanks also go to Chris George, UNU/IIST, for always being so helpful with issues related to the RSL-SAL tool.

References

1. Symbolic Analysis Laboratory, SAL (2001), http://sal.csl.sri.com
2. Bjørner, D.: New Results and Current Trends in Formal Techniques for the Development of Software for Transportation Systems. In: Proceedings of the Symposium on Formal Methods for Railway Operation and Control Systems (FORMS 2003), Budapest/Hungary, May 15-16. L'Harmattan Hongrie (2003)
3. Le Bliguet, M., Kjær, A.A.: Modelling Interlocking Systems for Railway Stations. Technical Report IMM-M.Sc.-2008-68, Informatics and Mathematical Modelling, Technical University of Denmark, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, Master thesis supervised by Anne Haxthausen (2008)
4. de Moura, L., Owre, S., Shankar, N.: The SAL Language Manual. Technical Report SRI-CSL-01-02, SRI International (2003), http://sal.csl.sri.com
5. Ehrig, H., Damm, W., Desel, J., Große-Rhode, M., Reif, W., Schnieder, E., Westkämper, E. (eds.): INT 2004. LNCS, vol. 3147, pp. 1–8. Springer, Heidelberg (2004)
6. Gjaldbæk, T., Haxthausen, A.E.: Modelling and Verification of Interlocking Systems for Railway Lines. In: Proceedings of the 10th IFAC Symposium on Control in Transportation Systems. Elsevier Science Ltd., Oxford (2003)
7. Lindegaard, M.P., Viuf, P., Haxthausen, A.E.: Modelling Railway Interlocking Systems. In: Proceedings of the 9th IFAC Symposium on Control in Transportation Systems 2000, Braunschweig, Germany, June 13-15, pp. 211–217 (2000)
8. Perna, J.I., George, C.: Model checking RAISE specifications. Technical Report 331, UNU-IIST, P.O.Box 3058, Macau (November 2006)
9. Perna, J.I., George, C.: Model Checking RAISE Applicative Specifications. In: Proceedings of the Fifth IEEE International Conference on Software Engineering and Formal Methods, pp. 257–268. IEEE Computer Society Press, Los Alamitos (2007)
10. The RAISE Language Group. The RAISE Specification Language. The BCS Practitioners Series. Prentice Hall Int., Reading (1992)
11. Schnieder, E., Tarnai, G. (eds.): Proceedings of Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2004), Braunschweig, Germany. Technical University of Braunschweig (December 2004)
12. Schnieder, E., Tarnai, G. (eds.): Proceedings of Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2007), Braunschweig, Germany. GZVB e.V (2007) ISBN 13:978-3-937655-09-3
13. Tarnai, G., Schnieder, E. (eds.): Proceedings of the Symposium on Formal Methods for Railway Operation and Control Systems (FORMS 2003), Budapest. L'Harmattan Hongrie (2003)

Refinement of Components in Connection-Safe Assemblies with Synchronous and Asynchronous Communication*

Rolf Hennicker[1], Stephan Janisch[1], and Alexander Knapp[2]

[1] Institut für Informatik, Ludwig-Maximilians-Universität München
{hennicker,janisch}@pst.ifi.lmu.de
[2] Institut für Informatik, Universität Augsburg
knapp@informatik.uni-augsburg.de

Abstract. Components are strongly encapsulated behaviours which interact with the environment by exchanging messages. Interaction may, amongst others, follow a synchronous rendezvous mechanism for message exchange or an asynchronous paradigm where sending and handling a message happens at different points in time. We extend our previously defined component model by integrating synchronous and asynchronous communication. As the formal background we use I/O-transition systems and consider asynchronous communication with fifo-ordered message buffers. We identify compatibility properties that should be satisfied when components communicate along synchronous and asynchronous connectors. As a first result we show that synchronous compatibility is a sufficient condition to ensure buffered compatibility in asynchronous communications. We introduce the notion of connection-safe assemblies which requires compatibility of both kinds of communication. We define a refinement relation and show its compositionality with respect to synchronous and asynchronous connectors in connection-safe assemblies. Finally, we provide results showing the preservation of connection-safety under component refinement.

1 Introduction

Structuring of large-scale software systems in terms of components and their interconnections is nowadays a standard in software development. Components can be characterised by the ability to encapsulate internal behaviour and to provide well-defined access points (often called ports) to the outside. This supports the construction of component assemblies, by connecting components via their ports, and the substitutability of components by relying only on their observable behaviour. Various software component models have been proposed; for an overview see [1] and for a comparison in the context of a "common component modelling example" (CoCoME) see [2].

The current paper sets out from our component model provided in [3] which has been equipped with a precise formalisation of the structural and behavioural aspects of components. This model distinguishes between simple (i.e., basic) components, component

* This research has been partially supported by the GLOWA-Danube project (01LW0602A2) sponsored by the German Federal Ministry of Education and Research.

C. Choppy and O. Sokolsky (Eds.): Monterey Workshop 2008, LNCS 6028, pp. 154–180, 2010.

assemblies, i.e., network structures of components connected via their ports, and composite components which encapsulate assemblies. For the formal representation of behaviours we use I/O-transition systems which are based on interface automata [4]. Explicit I/O-labellings allow us to distinguish between input, output and internal actions, which can be hidden to compute (derive) the observable behaviour of a component. The behaviour of an assembly is also an I/O-transition system which is derived from the composition of the observable behaviours of the components connected within the assembly. We assume that connections are always binary. For the computation of an assembly behaviour, the communication behaviour between connected components plays, of course, a central role. In [3] we have considered the case of synchronous communication, where communication is achieved by a rendezvous mechanism such that the sender and the receiver of a message synchronise on message exchange. This case is also considered in most other approaches like in ADLs such as Wright, Darwin or PADL [5]; but also software component models such as SOFA 2.0 and Fractal, both using behaviour protocols, or CoIn [6] with component interaction automata follow the synchronous communication scheme.

The first goal of this paper is to extend our component model in [3] by taking into account synchronous *and* asynchronous communication where a fifo-buffering mechanism is used when a message is sent along an asynchronous connector. Even though asynchronous communication with buffering is frequently used in practice, in particular in the context of distributed systems, its semantic properties are often neglected when it comes to behavioural analysis. One contribution of this paper lies in the rigorous formal treatment of asynchronous communication in our component model. For this purpose we provide a detailed definition for the computation of an assembly behaviour on the basis of synchronous and asynchronous connectors integrating explicit buffer behaviour and component behaviour on the level of I/O-transition systems. The resulting systems resemble communicating finite state machines (CFSM) with unbounded fifo-channels [7], and, in fact, we expect the theoretical results from the broad literature on the verification of CFSM systems, e.g., testing approaches to the unboundedness problem [8], to be more or less readily applicable to our semantics. In the realm of software component models, Maréchal et al. [9] give an approach to the analysis of components with asynchronous communication for Korrigan [10] based on symbolic transition systems. Asynchronous communication is explicitly taken into account by an integration of mailboxes for messages received but not yet processed; this work aims at developing algorithms for mailbox analysis, such as boundedness checks for mailboxes with (fifo) or without order (dictionary). Other approaches such as SOFA 2.0, Fractal, or Java/A [11] usually cope with asynchronous communication only in terms of an implementation which is not directly applicable to formal analysis. More recently, a Fractal extension [12] aims at a formal semantics for the behavioural modelling of distributed systems based on pNets [13]. However, the latter focuses on remote method calls as a mechanism for asynchronous communication.

None of the mentioned approaches provides equivalence or refinement relations for component behaviour, which directly leads to our second important goal, focusing on the study of component refinement and the investigation of properties concerning the communication behaviour of components (in assemblies) to be preserved under

refinements. To consider component refinement we must compare the observable behaviours of components which are given by I/O-transition systems. Since I/O-transition systems are based on the interface automata of de Alfaro and Henzinger [4], we can reuse their quite appealing ideas for refinement defined in terms of an alternating simulation relation. Essentially, any input of an abstract behaviour must be accepted as an input of the concrete behaviour (in related states) and, conversely, any output produced by a concrete behaviour must be producible as an output of the abstract behaviour (in related states). In principle, we adopt this idea but we suggest two extensions. First, we distinguish between internal actions and the invisible action τ, because internal actions naturally appear to express communications between components (which are neither input nor output actions) but which can not be abstracted away since we are interested in communication behaviours within component assemblies. Secondly, in contrast to [4], we require additional conditions for refinement, similar to the requirements of stuck-free conformance as defined in [14], to ensure that reactiveness of abstract behaviours is also valid in concrete behaviours.

On the basis of our refinement notion we study relationships between component refinement and communication behaviour exposed by component assemblies. For this purpose, we have to distinguish between synchronous and asynchronous communication. In the synchronous case two components (more precisely, observable component behaviours) are called *(synchronously) compatible* if any output issued by one component meets the other component in a state where this output will be accepted as an input, and vice versa. This notion, however, cannot be directly applied for asynchronously communicating components since then there is a delay between the action of sending a message to a buffer and the action of the target component taking the message from the buffer. Hence, in the asynchronous case, the interesting question is whether any message sent by one component will eventually be taken from the buffer by the other component for further processing. If this property is satisfied the two components are called *buffered compatible*. In our component model we allow both, synchronous and asynchronous connectors and we call a component assembly *connection-safe* if compatibility is ensured, for each kind of connectors, in any global system run.

As one major result we draw a connection between synchronous and asynchronous communication behaviour and show, by extending a theorem of [15], that synchronously compatible components are also buffered compatible if they are put in an asynchronous environment. From the practical point of view this result is rather relevant because checking buffered compatibility directly may soon become unmanageable while checks for synchronous compatibility are usually much easier. Our main theorems show that for closed assemblies with two connected components the following holds: First, the refinement relation is compositional for synchronous as well as for asynchronous component connectors and, secondly, the properties of synchronous and buffered compatibility, i.e. connection safety, are preserved by component refinement. In particular, this implies substitutability of components within an assembly by preserving connection-safety.

The paper is organised as follows: The basic definitions and facts for I/O-transition systems needed for this study are summarised in Sect. 2. In Sect. 3 we present the extension of our component metamodel to synchronous and asynchronous connectors and introduce a running example. In Sect. 4, we provide a detailed account on the

corresponding extension of the algebraic formalisation of our component model, integrating the definition of assembly behaviour with synchronous and asynchronous connectors. Sections 5 and 6 constitute our main results. In Sect. 5, the notion of connection-safe assembly is introduced and the relationship between the different compatibility notions is analysed. In Sect. 6 we define the refinement relation for I/O-transition systems and components and we provide the compositionality results for synchronous and asynchronous connectors. Finally, we give some concluding remarks in Sect. 7.

2 I/O-Transition Systems

We use I/O-transition systems to describe behaviours of ports, components, and assemblies with their provided (input) and required (output) operations as well as their internal actions. Our definition of I/O-transition system is similar to the notion of interface automata of de Alfaro and Henzinger [4]. However, we distinguish between internal actions and the *invisible* (or *silent*) action τ, because we are also interested in behaviours where internal actions should not be abstracted. For instance, we will focus on assembly behaviours of connected components where interactions between components are internal (because they are neither input nor output). Then we are interested in properties of interaction behaviours which can only be studied if internal actions are not abstracted. But, of course, when climbing up the hierarchal structure of components then the behaviour of a composite component, which encapsulates an assembly, will be obtained by abstracting the internal interactions to τ. In the following we summarise our notions for I/O-transition systems presented in [3] which will be used hereafter.

An *I/O-labelling*, *iol* for short, $L = (I, O, T)$ consists of three mutually disjoint sets of *input* labels I, *output* labels O, and *internal* labels T; we write $\bigcup L$ for the set of labels $I \cup O \cup T$. An *I/O-transition system*, *iots* for short, $A = (L, S, s_0, \Delta)$ is given by an iol L, a set of *states* S, an *initial state* $s_0 \in S$ and a *transition relation* $\Delta \subseteq S \times (\bigcup L \cup \{\tau\}) \times S$ (with $\tau \notin \bigcup L$). We write $L(A)$ for the iol of A.

2.1 Operators on I/O-Transition Systems

For deriving behaviours in our component framework we will use the following operators on iotss: hiding, relabelling and the formation of products. Hiding and relabelling on iotss are generalisations of the usual operators used in process algebras (see, e.g., [16,5]), and the product is defined in accordance with the product of interface automata [4].

Hiding. Hiding is used to turn a subset of the labels of an iots into the invisible action τ. Formally, the *hiding* of an iol $L = (I, O, T)$ w.r.t. a subset $H \subseteq \bigcup L$ is the iol $L/H = (I \setminus H, O \setminus H, T \setminus H)$. The *hiding* of an iots $A = (L, S, s_0, \Delta)$ w.r.t. a label set $H \subseteq \bigcup L$ is the iots $A/H = (L/H, S, s_0, \Delta/H)$ where $\Delta/H = \{(s, \tau, s') \mid (s, a, s') \in \Delta \wedge a \in H\} \cup \{(s, a, s') \mid (s, a, s') \in \Delta \wedge a \notin H\}$.

In some cases we will choose $H = T$, i.e., we will hide all internal labels. Then, for an iol $L = (I, O, T)$, we write $L\xi$ for L/T and for an iots A with $L(A) = (I, O, T)$, we write $A\xi$ for A/T.

Relabelling. A relabelling is used for renaming labels and for changing the kind of labels. Formally, a *relabelling* $\rho : L \to L'$ from an iol $L = (I, O, T)$ to an iol $L' = (I', O', T')$ is defined by a function from $\bigcup L$ to $\bigcup L'$ for which we also write ρ. The *relabelling* of an iots $A = (L, S, s_0, \Delta)$ w.r.t. a relabelling $\rho : L \to L'$ is the iots $A\rho = (L', S, s_0, \Delta\rho)$ where $\Delta\rho = \{(s, \rho(l), s') \mid (s, l, s') \in \Delta \wedge l \in \bigcup L\} \cup \{(s, \tau, s') \mid (s, \tau, s') \in \Delta\}$.

Given two relabellings $\rho_1 : L \to L'$ and $\rho_2 : L \to L'$, we define their union by $\rho_1 \cup \rho_2 : L \to L'$ with $(\rho_1 \cup \rho_2)(l) = \rho_1(l)$ if $\rho_2(l) = l$, $(\rho_1 \cup \rho_2)(l) = l$ otherwise.

Product. The formation of the product of two iotss expresses their parallel composition with synchronisation on identical input and output labels. To construct the product the iols of the given iotss must be composable. Two iolss $L_1 = (I_1, O_1, T_1)$ and $L_2 = (I_2, O_2, T_2)$ are *composable* if $I_1 \cap I_2 = \emptyset$, $O_1 \cap O_2 = \emptyset$, $T_1 \cap (I_2 \cup O_2 \cup T_2) = \emptyset$, and $T_2 \cap (I_1 \cup O_1 \cup T_1) = \emptyset$. The *shared* labels of composable iolss L_1 and L_2, written $L_1 \bowtie L_2$, are given by $(I_1 \cap O_2) \cup (O_1 \cap I_2)$. The *product* of two composable iolss L_1 and L_2 is the iol $L_1 \otimes L_2 = ((I_1 \cup I_2) \setminus (L_1 \bowtie L_2), (O_1 \cup O_2) \setminus (L_1 \bowtie L_2), T_1 \cup T_2 \cup (L_1 \bowtie L_2))$ which moves the shared labels to the internal labels. Two iotss A_1 and A_2 are *composable* if $L(A_1)$ and $L(A_2)$ are composable. The *product* of two composable iotss $A_1 = (L_1, S_1, s_{0,1}, \Delta_1)$ and $A_2 = (L_2, S_2, s_{0,2}, \Delta_2)$ is the iots $A_1 \otimes A_2 = (L_1 \otimes L_2, S_1 \times S_2, (s_{0,1}, s_{0,2}), \Delta)$ where

$$\begin{aligned}\Delta = &\{((s_1, s_2), a, (s_1', s_2)) \mid (s_1, a, s_1') \in \Delta_1 \wedge s_2 \in S_2 \wedge a \notin L_1 \bowtie L_2\} \cup \\ &\{((s_1, s_2), a, (s_1, s_2')) \mid (s_2, a, s_2') \in \Delta_2 \wedge s_1 \in S_1 \wedge a \notin L_1 \bowtie L_2\} \cup \\ &\{((s_1, s_2), a, (s_1', s_2')) \mid (s_1, a, s_1') \in \Delta_1 \wedge (s_2, a, s_2') \in \Delta_2 \wedge a \in L_1 \bowtie L_2\}\,.\end{aligned}$$

Pairwise composability for a set of iols implies that all composable pairs of this set have mutually disjoint shared labels. In the context of iots products, mutually disjoint shared labels guarantee that the synchronisation between different iotss is always binary. The product is commutative and associative. For a finite index set I, we write $\bigotimes_{i \in I} A_i$ for the product of the iotss A_i with $i \in I$.

3 Component Model with (A-)Synchronous Communication

We extend our component model presented in [3] to take into account not only synchronous but also asynchronous communication. By synchronous communication we understand a rendezvous mechanism where sender and receiver of a message synchronise on message exchange. In contrast, asynchronous communication works with fifo-buffering where the messages issued by a sender are buffered and can be taken (and processed) later on by the receiver. In our component model we distinguish between synchronous and asynchronous connectors which both are binary. For technical reasons we have considered in [3] also unary connectors, but apart from this point the component model described in the following is a conservative extension of the one in [3].

We consider components to be strongly encapsulated behaviours. Encapsulation is achieved by ports which regulate any interaction of components with their environment. Components can be hierarchically structured containing again an assembly of components and connectors. Figure 1 shows the metamodel of our component model. A *port*

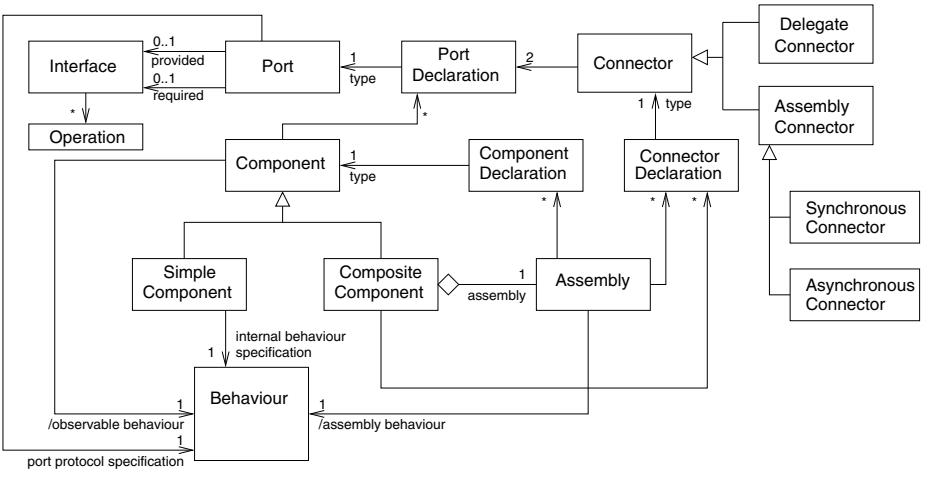


Fig. 1. Component metamodel

describes a view on a component. The operations offered by a port are summarised in its *provided interface*; the operations needed in its *required interface*. The sequencing of operation invocations issued and received by a port is described in a *port protocol specification*. To be precise, a port is in fact considered as a port type that can be used in local port declarations of a component.

There are two kinds of components, *simple components* and *composite components* which are abstracted in the metaclass *component*. Any kind of component has a set of *port declarations*, which introduce locally unique port names with corresponding port types, and an associated *observable behaviour* which describes the ordering of input and output actions of the component. In our metamodel a component represents in fact a component type that can be used in component declarations when building component assemblies. Each component should be correct with respect to its ports, i.e., the protocol of its ports should indeed be supported by the observable behaviour of the component. This correctness issue has been studied in [3]. For each simple component an *internal behaviour specification* is given which involves not only input and output actions but also transitions with internal actions. A composite component encapsulates an assembly of components. An *assembly* defines the internal structure of the composite component in terms of a set of local component declarations and local (binary) *assembly connector declarations* that connect local components via their ports. Assembly connectors can be synchronous or asynchronous. In a composite component, non-connected (open) ports of local components may be connected to so-called *relay ports* of the composite component, using *delegate connector declarations*. Also an assembly has an associated behaviour. As indicated by the slash symbol in Fig. 1 the observable behaviour of a component as well as the assembly behaviour are derived behaviours. The observable behaviour of simple components is derived from the components' given internal behaviour specification; for composite components the observable behaviour is derived from the behaviour of its assembly which in turn is derived from the observable

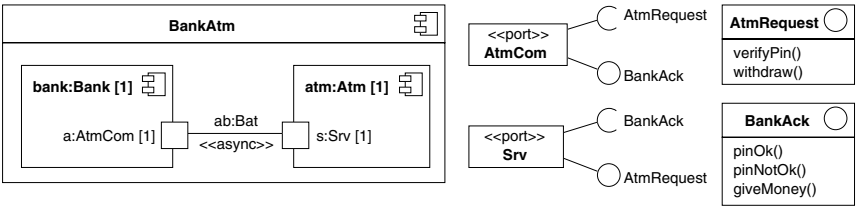


Fig. 2. Static structure of a simple Bank–Atm application

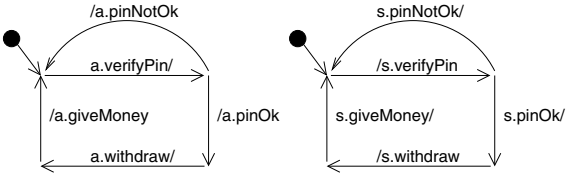


Fig. 3. Observable behaviour of Bank (left) and Atm (right)

behaviours of the local components within the assembly and their connections. In this paper we will particularly focus on assembly behaviours which depends on the synchronisation mechanisms used for the connectors.

Example 1 (Static structure and behaviours). Consider the simple Bank–Atm application in Fig. 2. A composite component type BankAtm contains an assembly of two simple components with types Bank and Atm introduced by the component declarations bank : Bank and atm : Atm.[1] The simple component types Bank and Atm have port declarations a : AtmCom and s : Srv resp. which are connected with an asynchronous assembly connector with name ab and type Bat.[2] The two simple component declarations and the connector form an assembly. The provided and required interfaces of the port types with their operations are depicted with the UML ball-and-socket notation on the right-hand side of Fig. 2. We do not consider operations with parameters here. If two ports are connected by an assembly connector, the provided interface of the one port has to be equal to the required interface of the other, and vice versa.[3]

Concerning behaviours we do not show port protocols and internal behaviour specifications of the simple component types Bank and Atm, but only give their derived observable behaviours in Fig. 3. Input and output messages are indicated by $p.m/$ and $/p.m$, respectively, where p is the port name on which the message is sent or received. Figure 4 shows the assembly behaviour of the asynchronously communicating Bank

[1] The UML2 declarations in Fig. 2 also show multiplicities, indicating how many instances of a component or port may exist. However, we only specified singletons (multiplicity 1) leaving the discussion of arbitrary multiplicities to future work.

[2] UML2 would allow for arbitrary n-ary connectors with $n > 2$ which we do not consider here.

[3] In general, one could use a more flexible condition such that the required interface of one port is included in the provided interface of the other one. However, it is technically more convenient to use the more restrictive condition from above.

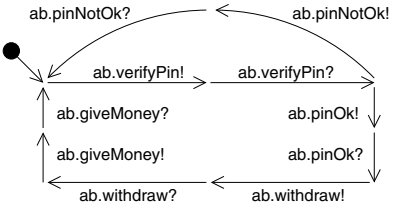


Fig. 4. Assembly behaviour of the Bank–Atm assembly

and Atm components. In this case a buffering behaviour of the connector ab is involved. Labels of the form ab.$m!$ represent the action of sending a message m on the connector ab which, at the same time, will be put into the input buffer on the opposite side of the connector. Labels of the form ab.$m?$ represent the action of taking a message out of the buffer. Taking out a message of the buffer semantically corresponds to the input of a message at a port as indicated in the observable behaviour of a component.

4 Formalisation of the Component Model

We will now provide a precise formalisation of our concepts for the static structure and for the dynamic behaviour of components with synchronous and asynchronous communication. For this purpose we complement the metamodel presentation of our component model with an algebraic description, which defines formally all previously mentioned concepts and behaviours in terms of algebraic structures and iotss resp.; see Sect. 2. In particular, we distinguish between those behaviours which have to be provided by the component developer and those that are computed (derived), by rendering the latter as definitions. We use *italics* to denote all kinds of derived operators. This section extends our formalisation in [3] to the asynchronous case.

4.1 Technical Prerequisites

Buffered I/O-Transition Systems. For asynchronous communication we need a mechanism for buffering (queueing) of messages. Technically, we use for this purpose buffered I/O-transition systems which model queues over a given set M of messages such that enqueueing a message m is an input action of the form $m!$ and dequeueing a message m is an output action of the form $m?$. The contents of a queue define the queue's states which are formally represented by the set M^* of finite sequences over M. The empty sequence is denoted by ϵ, the extension of $s \in M^*$ by an $m \in M$ at the front is denoted by $m \cdot s$, extension of the back end by $s \cdot m$.

Definition 1 (Queue iots). *Let M be a set. The* queue iots *over M is given by $Q^{\triangleleft}_M = ((I, O, T), S, s_0, \Delta)$, where $I = \{m! \mid m \in M\}$, $O = \{m? \mid m \in M\}$, $T = \emptyset$; $S = \{s \mid s \in M^*\}$, $s_0 = \epsilon$; and $\Delta \subseteq S \times (I \cup O \cup T \cup \{\tau\}) \times S$ is the smallest relation such that*

1. *for all* $m! \in I$ *and all* $s \in S$, *there exists* $(s, m!, s \cdot m) \in \Delta$,
2. *for all* $m? \in O$ *and all* $m \cdot s \in S$, *there exists* $(m \cdot s, m?, s) \in \Delta$.

For an iol (I, O, T) and a set of labels M we define the sets of labels $I_{M?} = \{l? \mid l \in I \cap M\} \cup (I \setminus M)$, $O_{M!} = \{l! \mid l \in O \cap M\} \cup (O \setminus M)$ and analogously for $I_{M!}$ and $O_{M?}$. If $I \subseteq M$ or $O \subseteq M$ we write $I?$, $I!$ or $O!$, $O?$ respectively. The *relabelling for buffered communication* $\beta_M : (I, O, T) \to (I_{M?}, O_{M!}, T)$ is defined by $\beta_M(l) = l?$ if $l \in I \cap M$, $\beta_M(l) = l!$ if $l \in O \cap M$, and $\beta_M(l) = l$ otherwise.

Specialised Relabellings. For the computation of the behaviours of components and assemblies we employ several relabelling functions, which specialise the relabelling introduced in Sect. 2.1. These relabellings are needed for creating shared labels when I/O-transition systems, representing behaviours, are composed. Even for asynchronous compositions shared labels will be needed for appropriate synchronisations with queue labels. We assume a primitive domain Nm of *names*.

A prefix relabelling prefixes all labels in an iots by a given name. For an iol $L = (I, O, T)$ and some name $n \in \mathrm{Nm}$, we define the iol $n.L = (n.I, n.O, n.T)$ where $n.I = \{n.i \mid i \in I\}$ and similarly for $n.O$ and $n.T$. The *prefix relabelling* $\rho_n : L \to n.L$ is defined by $\rho_n(l) = n.l$ for $l \in \bigcup L$. Given an iots A and a name $n \in \mathrm{Nm}$, we write $n.A$ for the iots $A\rho_n$.

A match relabelling maps differently prefixed labels to labels with a single common prefix. For an iol $L = (I, O, T)$, $X \subseteq \mathrm{Nm}$ and $y \in \mathrm{Nm}$, we define the iol $L\mu_{(X,y)} = (I\mu_{(X,y)}, O\mu_{(X,y)}, T\mu_{(X,y)})$ where $I\mu_{(X,y)} = \{y.l \mid \exists x \in X \,.\, x.l \in I\} \cup \{l \mid l \in I \wedge \forall x \in X \,.\, l \neq x.l'\}$ and analogously for $O\mu_{(X,y)}$ and $T\mu_{(X,y)}$. The *match relabelling* $\mu_{(X,y)} : L \to L\mu_{(X,y)}$ is defined by $\mu_{(X,y)}(x.l) = y.l$ if $x \in X$ and $x.l \in \bigcup L$, and $\mu_{(X,y)}(l') = l'$ otherwise.

For an iol $L = (I, O, T)$, $X \subseteq \mathrm{Nm}$ and $y \in \mathrm{Nm}$, a (binary) *synchronisation relabelling* $\sigma_{(X,y)}$ is given by a match relabelling $\mu_{(X,y)}$ with $|X| = 2$ and $T = T\mu_{(X,y)}$. An *asynchronous relabelling* $\alpha_{(X,y)}$ is given by the composition $\sigma_{(X,y)} \circ \beta_M$ of a relabelling β_M for buffered communication (cf. above) with $M = \{x.l \in I \cup O \mid x \in X, l \in \mathrm{Nm}\}$ and a synchronisation relabelling $\sigma_{(X,y)}$. Finally, a *relay relabelling* $\rho_{(x,y)}$ is given by a match relabelling $\mu_{(X,y)}$ with $X = \{x\}$.

4.2 Formalisation of Ports and Connectors

Ports. For the formalisation of ports we assume a domain Port of ports (more precisely, port types), a domain If of interfaces and a domain Msg of messages, together with functions $\mathrm{msg} : \mathrm{If} \to \wp\mathrm{Msg}$ to return the messages constructed from the operations of an interface, and $\mathrm{prv} : \mathrm{Port} \to \mathrm{If}$ and $\mathrm{req} : \mathrm{Port} \to \mathrm{If}$ for the provided and required interfaces of a port such that for all $P \in \mathrm{Port}$, $\mathrm{msg}(\mathrm{prv}(P)) \cap \mathrm{msg}(\mathrm{req}(P)) = \emptyset$. For a port P we write $msg(P)$ for $\mathrm{msg}(\mathrm{prv}(P)) \cup \mathrm{msg}(\mathrm{req}(P))$. We also assume a domain of port declarations $\mathrm{PortDcl}$ with a function $\mathrm{nm} : \mathrm{PortDcl} \to \mathrm{Nm}$ for the name and a function $\mathrm{ty} : \mathrm{PortDcl} \to \mathrm{Port}$ for the port (type); we write $p : P$ for a port declaration d with $\mathrm{nm}(d) = p$ and $\mathrm{ty}(d) = P$.

If ports are used for asynchronously communicating components, a queue iots is defined with respect to the messages provided at the particular port.

Definition 2 (Queue of a port). *The* queue *of a port (type)* P *is given by* $que(P) = Q^{\triangleleft}_{\mathrm{msg}(\mathrm{prv}(P))}$.

For each port $P \in \mathrm{Port}$ we assume given its *protocol (specification)*, written $\mathrm{prot}(P)$, which is an I/O-transition system $((I, O, T), Q, q_0, \Delta)$ with $I = \mathrm{msg}(\mathrm{prv}(P))$, $O = \mathrm{msg}(\mathrm{req}(P))$ and $T = \emptyset$.

Connectors. For building component assemblies and composite components we will connect port declarations of components. We assume a domain Conn of connectors (more precisely, connector types) with a function $\mathrm{ports} : \mathrm{Conn} \to \wp\mathrm{PortDcl}$ yielding the connected port declarations such that $|\mathrm{ports}(K)| = 2$ for each $K \in \mathrm{Conn}$, i.e. we consider binary connectors. We assume a domain of connector declarations $\mathrm{ConnDcl}$ with a function $\mathrm{nm} : \mathrm{ConnDcl} \to \mathrm{Nm}$ for the name and $\mathrm{ty} : \mathrm{ConnDcl} \to \mathrm{Conn}$ for the connector (type); we write $k : K$ for a connector declaration d with $\mathrm{nm}(d) = k$ and $\mathrm{ty}(d) = K$ and we write $k : (p : P, q : Q)$ if $\mathrm{ports}(K) = \{p : P, q : Q\}$.

The domain Conn has two disjoint sub-domains $\mathrm{AsmConn} \subseteq \mathrm{Conn}$, $\mathrm{DlgConn} \subseteq \mathrm{Conn}$ of assembly and delegate connectors, resp. Assembly connectors are used to connect port declarations of components when building up a component assembly. For an assembly connector with port declarations $\{p_1 : P_1, p_2 : P_2\}$ the required interface $\mathrm{req}(P_1)$ has to be equal to the provided interface $\mathrm{prv}(P_2)$ and vice versa. There are again two disjoint sub-domains $\mathrm{AsynchConn} \subseteq \mathrm{AsmConn}$, $\mathrm{SynchConn} \subseteq \mathrm{Conn}$ for asynchronous and synchronous connectors, resp.

Delegate connectors are used to connect open ports of an assembly with the relay ports of a surrounding composite component. For a delegate connector the provided and required interfaces of its port declarations must coincide.

Asynchronous connectors are used for asynchronous communication between the ports of components. Hence, they must show a buffering behaviour on each end of the connector in accordance with the messages that can be received (i.e. are provided) at a particular port.

Definition 3 (Buffering connector behaviour). *The* buffering behaviour *of an asynchronous connector* $k : (p : P, q : Q)$ *is given by* $buf(k : (p : P, q : Q)) = k.(que(P) \otimes que(Q))$.

To obtain a uniform definition of assembly behaviour below (Def. 4) we need for technical reasons a notion of "empty iots" which acts as a neutral element w.r.t. the product of iotss. Define $\mathbf{1}$ to be an iots (L, S, s_0, Δ) with $\bigcup L = \emptyset$, $S = \{s_0\}$ and $\Delta = \emptyset$. If a connector $k : K$ is synchronous we set $buf(k : K) = \mathbf{1}$.

4.3 Formalisation of Components and Assemblies

Components. We assume a domain Cmp of components (more precisely, component types) and a function $\mathrm{ports} : \mathrm{Cmp} \to \wp\mathrm{PortDcl}$ returning the ports declared for a component. For a component C and port declaration $p : P$ we write $C[p : P]$ to indicate that $p : P \in \mathrm{ports}(C)$. The port names p used in port declarations $p : P$ of one component C must be unique (but this is not necessary for port types P). Like for ports and connectors, we assume a domain of component declarations CmpDcl with a

function $\mathrm{nm} : \mathrm{CmpDcl} \to \mathrm{Nm}$ for the name and a function $\mathrm{ty} : \mathrm{CmpDcl} \to \mathrm{Cmp}$ for the component (type); we write $c : C$ for a component declaration d with $\mathrm{nm}(d) = c$ and $\mathrm{ty}(d) = C$. The ports of a component declaration are given by $\mathit{ports}(c : C) = \{c.p : P \mid p : P \in \mathrm{ports}(C)\}$.

For each component (type) $C \in \mathrm{Cmp}$ there is a derived *observable behaviour*, written $\mathit{obs}(C)$, which is an iots $((I, O, T), Q, q_0, \Delta)$ with $I = \bigcup\{p.\mathrm{msg}(\mathrm{prv}(P)) \mid p : P \in \mathrm{ports}(C)\}$, $O = \bigcup\{p.\mathrm{msg}(\mathrm{req}(P)) \mid p : P \in \mathrm{ports}(C)\}$ and $T = \emptyset$. Hence, the observable labels of a component (type) are just the labels according to the (provided and required) messages of the ports of the component prefixed with the port name in the corresponding port declaration. The only additional action that can occur in the observable behaviour of a component is the invisible action τ. As already indicated in the component metamodel in Fig. 1 the observable behaviour of a component is a derived behaviour. Its definition depends on whether the component is simple or composite; cf. Def. 5 and Def. 6 below. The *observable behaviour* of a component declaration $c : C$ is given by $\mathit{obs}(c : C) = c.\mathit{obs}(C)$.

Assemblies. Let us now formalise the static structure and the behaviour of component assemblies. An assembly contains a set of component declarations and a set of connector declarations which connect ports (more precisely, the connector declarations connect port declarations belonging to component declarations of the assembly). We assume a domain Asm of assemblies with functions $\mathrm{cmps} : \mathrm{Asm} \to \wp\mathrm{CmpDcl}$ returning an assembly's declared components and $\mathrm{conns} : \mathrm{Asm} \to \wp\mathrm{ConnDcl}$ yielding its declared connectors. The component names c used in component declarations $c : C$ of an assembly a must be unique (but this is not necessary for the component types C). Similarly, connector names within the assembly must be unique. For an assembly a we define the subset of asynchronous connectors by $\mathit{acs}(a) \subseteq \mathrm{conns}(a)$ such that $k : K \in \mathit{acs}(a)$ iff $K \in \mathrm{AsynchConn}$, and the subset of synchronous connectors by $\mathit{scs}(a) \subseteq \mathrm{conns}(a)$ such that $k : K \in \mathit{scs}(a)$ iff $K \in \mathrm{SynchConn}$.

An assembly $a \in \mathrm{Asm}$ has to be well-formed: (i) it shows only assembly connectors, i.e., if $k : K \in \mathrm{conns}(a)$, then $K \in \mathrm{AsmConn}$; (ii) only ports of components inside a are connected, i.e., for all $k : K \in \mathrm{conns}(a)$ we have that $\mathrm{ports}(K) \subseteq \bigcup\{\mathit{ports}(c : C) \mid c : C \in \mathrm{cmps}(a)\}$; and (iii) there is at most one connector for each port, i.e., if $c.p : P \in \bigcup\{\mathit{ports}(c : C) \mid c : C \in \mathrm{cmps}(a)\}$ and $k : K, k' : K' \in \mathrm{conns}(a)$ with $c.p : P \in \mathrm{ports}(K) \cap \mathrm{ports}(K')$, then $k : K = k' : K'$.

To retrieve component declarations from port declarations within an assembly a we define $\mathit{cmp} : \bigcup\{\mathit{ports}(c : C) \mid c : C \in \mathrm{cmps}(a)\} \to \mathrm{cmps}(a)$ by $\mathit{cmp}(c.p : P) = c : C$ if $c.p : P \in \mathit{ports}(c : C)$. The components of an assembly a may show *open* ports which are not connected and we let $\mathit{open}(a) = \bigcup\{\mathit{ports}(c : C) \mid c : C \in \mathrm{cmps}(a)\} \setminus \bigcup\{\mathrm{ports}(K) \mid k : K \in \mathrm{conns}(a)\}$.

Let us now focus on the definition of the behaviour of an assembly. The idea is that the behaviour of an assembly is determined by the composition of the observable behaviours of the components occurring in the assembly. But, of course, the composition must be defined in accordance with the possible communications between components which are connected via their ports. Since connectors may be asynchronous the buffering behaviour of connectors (cf. Def. 3) plays a crucial role. Moreover, some matching relabellings are necessary to achieve the desired behaviour.

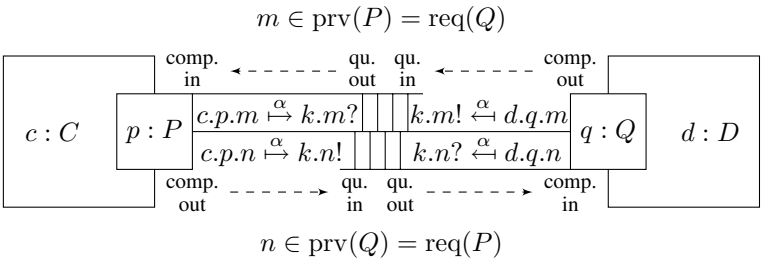


Fig. 5. Assembly with asynchronous connector

Figure 5 illustrates how the behaviour of an assembly with two asynchronously communicating components is constructed. There are two component declarations $c : C$ and $d : D$. The component type C has one port declaration $p : P$ which, in the context of the assembly with component declaration $c : C$, is considered as a port declaration $c.p : P$ to ensure uniqueness of port names within an assembly. Similarly, the component type D has one port declaration $q : Q$. Thus the messages sent out from the component c via its port p have the form $c.p.n$ where n is a message of the required interface of P. The two ports are connected by a connector declaration of the form $k : (c.p : P, d.q : Q)$. Thus the required interface of P must coincide with the provided interface of Q. According to the buffering behaviour of the connector k there is a queue $que(Q)$ (cf. Def. 2) which allows inputs of the form $k.n!$ with n being a message according to the provided interface of Q. To achieve that the issued message $c.p.n$ will indeed be put into the queue $que(Q)$, we use a matching relabelling α which maps $c.p.n$ to $k.n!$. The message n can be dequeued from $que(Q)$ later on with the action $k.n?$. Since the component d inputs on its port q messages of the form $d.q.n$ we use again the matching relabelling α which now maps $d.q.n$ to $k.n?$. The communication in the other direction works analogously.

Figure 6 illustrates how the behaviour of an assembly with two synchronously communicating components is constructed. Here, the necessary relabelling to synchronise input and output actions is much easier. Indeed, in this case a message $c.p.n$ sent from component c via its port p must be matched with the input action $d.q.n$ on the port q of the component d. For this purpose both actions are simply matched to the label $k.n$ with the relabelling σ, where k is again the connector's name.

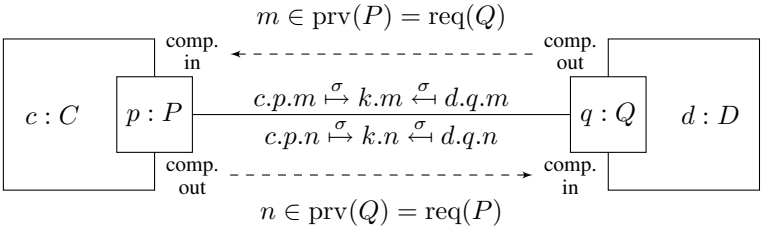


Fig. 6. Assembly with synchronous connector

For general assemblies, we apply the match relabellings defined in Sect. 4.1 to define the relabelling α for asynchronous and σ for synchronous connectors.

$$\alpha \equiv \bigcup\{\alpha_{(\{c.p,d.q\},k)} \mid k : K \in acs(a)\,.\,\mathrm{ports}(K) = \{c.p : P, d.q : Q\}\}\,,$$
$$\sigma \equiv \bigcup\{\sigma_{(\{c.p,d.q\},k)} \mid k : K \in scs(a)\,.\,\mathrm{ports}(K) = \{c.p : P, d.q : Q\}\}\,.$$

We have now the technical ingredients to define the behaviour of an assembly. In the definition the successive application of α and σ cannot lead to conflicts because both relabellings are disjoint. Note also that in the case of synchronous connectors $buf(k : K)$ is trivial as explained in Sect. 4.2. Moreover, the assembly behaviour is well-defined because all participating behaviours are composable. This is due to the disjointness of provided and required operations on port types, to the uniqueness of names for ports (in component declarations) as well as for components and connectors (in the assembly), and to the commutativity and associativity of the composition operator for iotss.

Definition 4 (Assembly behaviour). *The* behaviour *of an assembly a is given by*

$$beh(a) = \bigotimes\nolimits_{c:C\in\mathrm{cmps}(a)}(obs(c : C)\alpha\sigma) \otimes \bigotimes\nolimits_{k:K\in\mathrm{conns}(a)} buf(k : K)\,.$$

We write $\langle\mathcal{C};\mathcal{K}\rangle$ for an assembly a with the set of component declarations $\mathrm{cmps}(a) = \mathcal{C}$ and the set of connector declarations $\mathrm{conns}(a) = \mathcal{K}$.

Example 2 (Assembly behaviour). The static structure of the Bank–ATM application in Fig. 2 is formally represented by an assembly $\langle$bank : Bank, atm : Atm; ab : Bat$\rangle$. The assembly behaviour, shown in Fig. 4, is obtained from the composition of the observable behaviours of the components Bank and Atm with the buffering behaviour of the asynchronous connector Bat:

$$beh(\langle\mathsf{bank : Bank, atm : Atm; ab : Bat}\rangle) =$$
$$obs(\mathsf{bank : Bank})\alpha\sigma \otimes obs(\mathsf{atm : Atm})\alpha\sigma \otimes buf(\mathsf{ab : Bat})\,.$$

Simple Components. We assume a sub-domain $\mathrm{SCmp} \subseteq \mathrm{Cmp}$ of simple components. Each $SC \in \mathrm{SCmp}$ has a user defined *internal behaviour specification* $\mathrm{beh}(SC)$, which is an iots $((I, O, T), Q, q_0, \Delta)$ with $I = \{p.\mathrm{msg}(\mathrm{prv}(P)) \mid p : P \in \mathrm{ports}(SC)\}$ and $O = \{p.\mathrm{msg}(\mathrm{req}(P)) \mid p : P \in \mathrm{ports}(SC)\}$. The observable behaviour of a simple component SC is derived from its internal behaviour specification by hiding all internal labels. Technically this is achieved with the hiding operator ξ; see Sect. 2.

Definition 5 (Observable behaviour of simple component). *The* observable behaviour *of a simple component SC is given by $obs(SC) = \mathrm{beh}(SC)\xi$.*

Example 3 (Observable behaviours). The observable behaviour hides internal transitions, i.e. relabels internal transition to τ. In order to keep our running example simple and illustrative we refrained from modelling internal behaviour and assumed observable behaviours without τ transitions instead (cf. Fig. 3).

Composite Components. Composite components are constructed by encapsulating an assembly and by connecting, with delegate connectors, the open ports of the assembly with relay ports of the composite component. Formally, we assume a sub-domain

$\mathrm{CCmp} \subseteq \mathrm{Cmp}$ of composite components, disjoint to SCmp and functions $\mathrm{asm} : \mathrm{CCmp} \to \mathrm{Asm}$ returning the underlying assembly of a composite component, and $\mathrm{conns} : \mathrm{CCmp} \to \wp\mathrm{ConnDcl}$ returning the connectors declared in a composite component. Similar to assemblies we require a composite component CC to be well-formed: (i) it shows only delegate connectors, i.e., if $k : K \in \mathrm{conns}(CC)$, then $K \in \mathrm{DlgConn}$; (ii) all open ports of the $\mathrm{asm}(CC)$ are connected, i.e., for all $c.p : P \in \mathit{open}(\mathrm{asm}(CC))$ there is $k : K \in \mathrm{conns}(CC)$ such that $c.p : P \in \mathrm{ports}(K)$; and (iii) all relay ports are connected, i.e., for all $r : R \in \mathrm{ports}(CC)$ there is a unique $k : K \in \mathrm{conns}(CC)$ with $\mathrm{ports}(K) = \{c.p : P, r : R\}$ and $c.p : P \in \mathit{open}(\mathrm{asm}(CC))$.

The observable behaviour of a composite component is derived from the behaviour of its underlying assembly by hiding all internal actions, which in the case of assemblies are the communications on the connectors, and by matching the labels on the open ports of the assemblies with the labels on the relay ports in accordance with the delegate connectors.

Definition 6 (Observable behaviour of composite component). *The* observable behaviour *of a composite component CC is given by*

$$\mathit{obs}(CC) = (\mathit{beh}(\mathrm{asm}(CC))\xi)\rho \,,$$

where $\rho = \bigcup\{\rho_{(c.p,r)} \mid k : K \in \mathrm{conns}(CC) \,.\, \mathrm{ports}(K) = \{c.p : P, r : R\}\}$.

We write $\langle a; \mathcal{P}; \mathcal{K}\rangle$ for a composite component CC with assembly $\mathrm{asm}(CC) = a$, set of (relay) port declarations $\mathrm{ports}(CC) = \mathcal{P}$ and set of (delegate) connector declarations $\mathrm{conns}(CC) = \mathcal{K}$.

Example 4 (Observable behaviour of composite components). Since the Bank–Atm application is a closed system, the observable behaviour of the composite component $\langle\langle \mathsf{bank} : \mathsf{Bank}, \mathsf{atm} : \mathsf{Atm}; \mathsf{ab} : \mathsf{Bat}\rangle; \emptyset; \emptyset\rangle$ consists of τ-transitions only.

4.4 Buffered Components

The assembly behaviour has been defined on the basis of connectors which may show an asynchronous buffering behaviour. We show that the assembly behaviour can also be computed by rearranging the buffers in such a way that they do not belong to the connectors but to the components. For this purpose we introduce a new kind of component behaviour which integrates observable behaviour and message buffers for a single component based on a notion of buffered iotss.

An iots $A = (L, S, s_0, \Delta)$ is *without queue labels* if L does not contain labels of the form $m?$ or $m!$ (dequeue and enqueue labels of queue iotss; cf. Def. 1).

Lemma 1. *If $A = ((I, O, T), S, s_0, \Delta)$ is without queue labels, $X \subseteq I$ and $Y \subseteq O$, then $A\beta_{X\cup Y}$ and $Q_X^{\triangleleft}$ are composable.* □

The relabelling $\beta_{X\cup Y}$, defined in Sect. 4.1, prepares A on the one hand for the synchronisation with its queue $Q_X^{\triangleleft}$, and, on the other hand for synchronisation with an iots which provides matching inputs for the asynchronous outputs of A on Y.

Definition 7 (Buffered iots). *Let $A = ((I, O, T), S, s_0, \Delta)$ be an iots without queue labels. Let $X = \{X_1, \ldots, X_n\}$ where $X_i \subseteq I$ and $X_i \cap X_j = \emptyset$ for all $i \neq j \in \{1, \ldots, n\}$. The* buffered iots *for A with buffered input X and buffered output $Y \subseteq O$ is given by*

$$\Omega_{X,Y}(A) = A\beta_{X_1 \cup \cdots \cup X_n \cup Y} \otimes Q^{\triangleleft}_{X_1} \otimes \cdots \otimes Q^{\triangleleft}_{X_n} ,$$

If $X = I$ and $Y = O$, then the iots is completely buffered *and we write $\Omega(A)$.*

Lemma 2. *If $A = ((I, O, T), S, s_0, \Delta)$ is without queue labels, $X = \{X_1, \ldots, X_n\}$ where $X_i \subseteq I$ and $X_i \cap X_j = \emptyset$ for all $i \neq j \in \{1, \ldots, n\}$, and $Y \subseteq O$, then $L(\Omega_{X,Y}(A)) = (I_{(X_1 \cup \cdots \cup X_n)!}, O_{Y!}, T \cup (I_{(X_1 \cup \cdots \cup X_n)?} \setminus I))$.* □

Buffered iotss are later used to develop and analyse notions of refinement and compatibility on the level of iotss. The results are then applied to our formal component model. Therefore, we detail in the following on buffered iotss and their intended application as a formal representation of asynchronously communicating components.

Example 5 (Buffered iots). Buffered iotss for the observable behaviour of components are obtained from their composition with input queues defined w.r.t. the ports of the given component. For instance, the Bank component is equipped with one port only, therefore we have $\Omega_{X,Y}(obs(\mathsf{Bank})) = obs(\mathsf{Bank})\beta_{X \cup Y} \otimes Q^{\triangleleft}_X$, where $X = \{\{\mathsf{a}.m \mid m \in \{\mathsf{verifyPin}, \mathsf{withdraw}\}\}\}$ and $Y = \{\mathsf{a}.m \mid m \in \{\mathsf{pinOk}, \mathsf{pinNotOk}, \mathsf{giveMoney}\}\}$.

Definition 8 (Communication behaviour of component). *The* communication behaviour *of a component C buffered on a set of port declarations $\mathcal{P}$ is given by*

$$com_{\mathcal{P}}(C) = \Omega_{X,Y}(obs(C)) ,$$

where $X = \{X_{p:P} \mid p : P \in \mathcal{P} \cap \mathrm{ports}(C)\}$, $X_{p:P} = \{p.m \mid m \in \mathrm{msg}(\mathrm{prv}(P))\}$, $Y = \{p.m \mid p : P \in \mathcal{P} \cap \mathrm{ports}(C) \wedge m \in \mathrm{msg}(\mathrm{req}(P))\}$.

The communication behaviour of a component declaration $c : C$ w.r.t. a set of port declarations $\mathcal{P} \subseteq \mathrm{ports}(C)$ is given by $com_{\mathcal{P}}(c : C) = c.com_{\mathcal{P}}(C)$.

In order to obtain a characterisation of assembly behaviour in terms of communicating buffered components we have to ensure commutativity and associativity of compositions of buffered iotss. Concerning composability we record only the special case of completely buffered iotss, which is later needed in our analysis.

Lemma 3. *Let A and B be iotss without queue labels. If A and B are composable, then $\Omega(A)$ and $\Omega(B)$ are composable.* □

The lemma holds also for arbitrary buffered iotss, if we ensure that the input partitions determining the asynchronous input of one iots is consistent with the relabelling for asynchronous output of its communication partner. As a consequence the composition of buffered iotss is associative and commutative. The proof of these facts is tedious but rather straightforward.

Definition 8 includes the synchronisation of the observable behaviour of a component with its input queues. By composition of communication behaviours, we synchronise output transitions of one component behaviour with input transitions of the

queues of other components. Hence we can obtain an assembly behaviour by composition of communication behaviours of components which is equivalent to Def. 4. For taking into account the names of asynchronous connectors, we replace the asynchronous relabelling α used in Def. 4 by a slightly simpler relabelling, defined by $\kappa \equiv \bigcup\{\mu_{(\{c.p,d.q\},k)} \mid k : K \in acs(a)\,.\,\mathrm{ports}(K) = \{c.p : P, d.q : Q\}\}$.

Proposition 1. *If a is an assembly, then $beh(a) = \bigotimes_{c:C\in\mathrm{cmps}(a)}(com_{\mathcal{P}}(c : C)\kappa\sigma)$, where $\mathcal{P} = \bigcup\{\mathrm{ports}(K) \mid k : K \in acs(a)\}$.* □

5 Connection-Safe Assemblies

A safe communication of two components over a synchronous connector is characterised by the fact that if one component is about to send a message the other component is indeed willing to accept this message. For the iotss underlying the components this means that if the one iots has reached a state where it does an output, the other iots is in a state where it does the corresponding input. This idea of synchronous safe communication is captured by the following notion of compatibility of iotss as introduced by Gouda, Manning, and Yu for communicating finite state machines [17] and used by de Alfaro and Henzinger for iotss [4], which is based on the reachable states of iotss: The *reachable states* $\mathscr{R}(A)$ of an iots $A = (L, S, s_0, \Delta)$ are inductively defined as follows: $s_0 \in \mathscr{R}(A)$; and if $s \in \mathscr{R}(A)$ and there is an $a \in \bigcup L \cup \{\tau\}$ and an $s' \in S$ with $(s, a, s') \in \Delta$, then $s' \in \mathscr{R}(A)$.

Definition 9 (Compatibility). *Let $A = ((I_A, O_A, T_A), S_A, s_{0,A}, \Delta_A)$ and $B = ((I_B, O_B, T_B), S_B, s_{0,B}, \Delta_B)$ be composable iotss. B is a* compatible context *for A, if for all $l \in O_A \cap I_B$ and all $(s_A, s_B) \in \mathscr{R}(A \otimes B)$, if $(s_A, l, s'_A) \in \Delta_A$, then there exists $(s_B, l, s'_B) \in \Delta_B$. The iotss A and B are* compatible *if A is a compatible context for B and vice versa.*

Example 6 (Compatible iotss). The iotss representing the observable behaviours of the components Bank and Atm in Fig. 3 are obviously compatible. All outputs are immediately synchronised in both directions (modulo port relabelling).

For asynchronously communicating components which are connected by buffers the situation of safe communication is different: We have to ensure that each message sent out by one component is eventually understood by the receiving component. For technical reasons we restrict our attention to infinite communication sequences. Then safe communication means that if we observe an infinite communication sequence with output labels putting a message into a buffer and input labels taking a message from a buffer we have to be able to pair off the corresponding output and input labels. We thus base the notion of buffered compatibility of buffered iotss as an analogue to (synchronous) compatibility of iotss on infinite label sequences and require for buffered compatibility a pairing function for sending and taking. An *infinite run* of an iots $A = (L, S, s_0, \Delta)$ is an infinite sequence $s_0, l_0, s_1, l_1, \ldots$ with $s_n \in S$, $l_n \in \bigcup L$, and $(s_n, l_n, s_{n+1}) \in \hat{\Delta}$ for all $n \in \mathbb{N}$. An *infinite weak trace* of A is a sequence $l_0, l_1, \ldots$ with $l_n \in \bigcup L$ such that there is a run $s_0, l_0, s_1, l_1, \ldots$ of A.

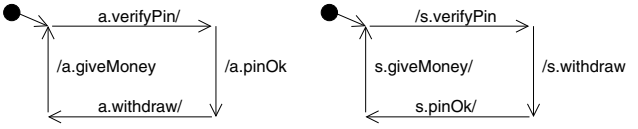


Fig. 7. Example for buffered compatible iotss

Definition 10 (Buffered compatibility). *Let (L, S, s_0, Δ) be an iots and $\mu : U \to V$ a mapping with $U, V \subseteq \bigcup L$. An infinite weak trace $l_0, l_1, \ldots$ of (L, S, s_0, Δ) is* μ-buffered compatible*, if for each $u \in U$ there is a bijection $\varphi_u : \{k \in \mathbb{N} \mid l_k = u\} \to \{k \in \mathbb{N} \mid l_k = \mu(u)\}$ with $k < \varphi_u(k)$.*

Let A and B be composable iotss without queue labels and $L(A) = (I_A, O_A, T_A)$ and $L(B) = (I_B, O_B, T_B)$. Let $\mu : \{m! \mid m \in O_A \cup O_B\} \to \{m? \mid m \in I_A \cup I_B\}$ be defined by $\mu(m!) = m?$. An infinite weak trace of $\Omega(A) \otimes \Omega(B)$ is buffered compatible*, if it is μ-buffered compatible. A and B are* buffered compatible*, if all infinite weak traces of $\Omega(A) \otimes \Omega(B)$ are buffered compatible.*

Example 7 (Buffered compatible iotss). Asynchronous communication allows for simultaneous sending of messages as illustrated, for instance, in the iotss of Fig. 7. Compared to the behaviours known from Fig. 3, the order of s.withdraw and s.pinOk in the right-hand iots was swapped and both iots were reduced to one path. The composition of the corresponding buffered iotss results in an iots where the possibility of simultaneous sending is modelled by the respective queue actions (cf. Fig. 8).

Obviously the iotss in Fig. 7 are not synchronously compatible, due to the output of the messages a.pinOk and s.withdraw. However, they are buffered compatible. Figure 8 shows the product iots, which would be obtained along appropriate relabellings (cf. Prop. 1) and an asynchronous connector ab : Bat as above. The infinite weak traces of the product allow to match the enqueue actions ab.verifyPin!, ab.withdraw!, etc. with their

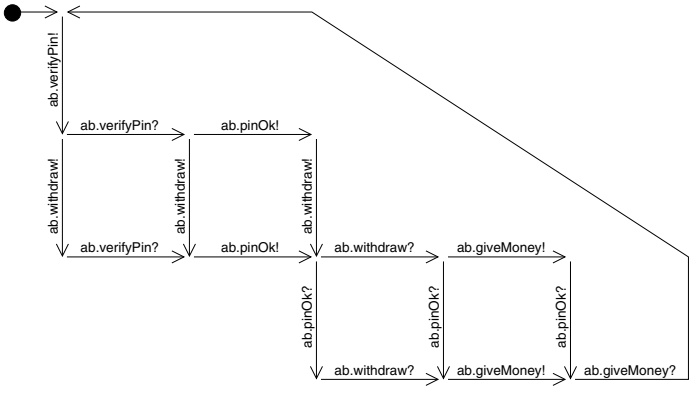


Fig. 8. Product of buffered iotss with simultaneous sending

dequeuing counterparts ab.verifyPin?, ab.withdraw?, etc. as required by the definition of μ-buffered compatibility.

In the context of component assemblies, compatibility of two synchronously or asynchronously communicating iotss has to be lifted to an arbitrary number of communicating components. We therefore introduce the notion of connection-safe assemblies.

Definition 11 (Connection-safety). *An assembly* $a = \langle c_1 : C_1, \ldots, c_n : C_n; \mathcal{K} \rangle$ *is* connection-safe*, if for all connector declarations* $k : (c_i.p : P, c_j.q : Q) \in \mathcal{K}$ *the following conditions hold:*

1. *If* k *is synchronous, let* $obs(c_i : C_i) = (L_{c_i}, S_{c_i}, s_{0,c_i}, \Delta_{c_i})$ *and* $obs(c_j : C_j) = (L_{c_j}, S_{c_j}, s_{0,c_j}, \Delta_{c_j})$*. Then for all* $(s_{c_1}, \ldots, s_{c_n}, K) \in \mathscr{R}(\mathrm{beh}(a))$*, if* $(s_{c_i}, c_i.p.m, s'_{c_i}) \in \Delta_{c_i}$ *for some* $m \in \mathrm{msg}(\mathrm{req}(P))$*, then there is a* $(s_{c_j}, c_j.q.m, s'_{c_j}) \in \Delta_{c_j}$*; and if* $(s_{c_j}, c_j.q.m, s'_{c_j}) \in \Delta_{c_j}$ *for some* $m \in \mathrm{msg}(\mathrm{req}(Q))$*, then there is a* $(s_{c_i}, c_i.p.m, s'_{c_i}) \in \Delta_{c_i}$*.*
2. *If* k *is asynchronous, let* $\mu : \{k.m! \mid m \in \mathrm{msg}(\mathrm{req}(P)) \cup \mathrm{msg}(\mathrm{req}(Q))\} \to \{k.m? \mid m \in \mathrm{msg}(\mathrm{prv}(Q)) \cup \mathrm{msg}(\mathrm{prv}(P))\}$ *with* $\mu(k.m!) = k.m?$*. Then all infinite weak traces of* $\mathrm{beh}(a)$ *are* μ*-buffered compatible.*

Note that for assemblies consisting of just two components with one port each and being connected by a single either synchronous or asynchronous connector, connection safety just means (synchronous) compatibility or buffered compatibility of the iotss underlying the communication behaviour of components of Def. 8.

The different concepts of compatibility for rendezvous and buffered communication raise the question whether synchronous compatibility of two iotss induces their buffered compatibility when they are put into an asynchronous context; we concentrate on closed compositions, where an iots $((I, O, T), S, s_0, \Delta)$ is *closed* if $I = O = \emptyset$. In order to answer this question, we first extend a result by Cécé and Finkel [15, Thm. 35] that, under some restrictions, two compatible finite state machines yield, when communicating through queues, a so-called half-duplex system. In our context of iotss, a composition $\Omega(A) \otimes \Omega(B)$ is *half-duplex*, if in every reachable state $((s_A, q_A), (s_B, q_B)) \in \mathscr{R}(\Omega(A) \otimes \Omega(B))$ one of the queues is empty: $q_A = \epsilon \vee q_B = \epsilon$. The proof of this result was by contradiction and used the restriction that the two compatible communicating finite state machines are deterministic and have no so-called mixed states, i.e., states where both an input and an output can happen. For iotss, this corresponds to input separation, that is, for each state showing some outgoing input transition all outgoing transitions are labelled by inputs: An iots $A = (L, S, s_0, \Delta)$ with $L = (I, O, T)$ is *input separated*, if for all $s \in \mathscr{R}(A)$ with $(s, l, s') \in \Delta$ for some $s' \in S$ and $l \in I$, then $\{a \in \bigcup L \cup \{\tau\} \mid \exists s' \in S \,.\, (s, a, s') \in \Delta\} \subseteq I$. For example the iotss in Fig. 3 are input separated, since all states with outgoing input transitions show only input transitions. Input separation may be understood as a property which reflects single-threaded execution: output and internal transitions succeeding an input are considered to encode the reaction of the component to this input.

Moreover, the restriction to deterministic iotss would not be appropriate for our setting because in the context of invisible actions non-determinism arises quite naturally. After establishing the result that compatible, input separated iots A and B induce a half-duplex asynchronous system $\Omega(A) \otimes \Omega(B)$ by a direct proof based on an invariant, we

show that if we additionally require A and B to always eventually take an input, then A and B are buffered compatible. An iots $A = (L, S, s_0, \Delta)$ with $L = (I, O, T)$ is *always eventually inputting*, if all weak infinite traces $l_0, l_1, \ldots$ show infinitely many $l_n \in I$.

Lemma 4. *Let A and B be composable, input separated iotss without queue labels, and let $A \otimes B$ be closed. If A and B are compatible, then $\Omega(A) \otimes \Omega(B)$ is half-duplex.*

Proof. Let us first fix some terminology: For an iots (L, S, s_0, Δ) the *transitive closure* of Δ is the relation $\Delta^* \subseteq S \times \bigcup L^* \times S$ defined inductively as follows: $(s, \epsilon, s') \in \Delta^*$, if $s = s'$; $(s, l \cdot \lambda, s') \in \Delta^*$, if there is an $s'' \in S$ with $(s, l, s'') \in \Delta$, and $(s'', \lambda, s') \in \Delta^*$. The *transitive τ-closure* of Δ is the relation $\hat{\Delta}^* \subseteq S \times \bigcup L^* \times S$ defined inductively by: $(s, \epsilon, s') \in \hat{\Delta}^*$, if $(s, \tau, s') \in \hat{\Delta}$; $(s, l \cdot \lambda, s') \in \hat{\Delta}^*$, if there is an $s'' \in S$ with $(s, l, s'') \in \hat{\Delta}$, and $(s'', \lambda, s') \in \hat{\Delta}^*$. The *safe label sequences* $\Delta^*(s) \subseteq \bigcup L^*$ in a state $s \in S$ are inductively given by: $\epsilon \in \Delta^*(s)$ for all $s \in S$; $l \cdot \lambda \in \Delta^*(s)$, if there is an $s' \in S$ with $(s, l, s') \in \Delta$, and for all $s' \in S$ with $(s, l, s') \in \Delta$ it holds that $\lambda \in \Delta^*(s')$. For an iots $A = ((I, O, T), S, s_0, \Delta)$ we write I_A for I, O_A for O and similarly for the other parts.

Define, using the hiding operator ξ defined in Sect. 2.1,

$$
\begin{aligned}
R = \{(((s_A, q_A), (s_B, q_B)), (r_A, r_B)) \mid \\
((s_A, q_A), (s_B, q_B)) \in \mathscr{R}(\Omega(A) \otimes \Omega(B)) \wedge (r_A, r_B) \in \mathscr{R}(A \otimes B) \wedge \\
((q_A = \epsilon \wedge q_B = \epsilon \wedge (s_A, s_B) = (r_A, r_B)) \vee \\
(q_A = \epsilon \wedge q_B \neq \epsilon \wedge s_B = r_B \wedge (r_A, q_B, s_A) \in \hat{\Delta}^*_{A\xi} \wedge q_B \in \Delta^*_B(r_B) \vee \\
(q_A \neq \epsilon \wedge q_B = \epsilon \wedge s_A = r_A \wedge (r_B, q_A, s_B) \in \hat{\Delta}^*_{B\xi} \wedge q_A \in \Delta^*_A(r_A)))\}
\end{aligned}
$$

We show that for all reachable $((s_A, q_A), (s_B, q_B)) \in \mathscr{R}(\Omega(A) \otimes \Omega(B))$ it holds that $\exists (r_A, r_B) \,.\, (((s_A, q_A), (s_B, q_B)), (r_A, r_B)) \in R$. Then, by definition of R, always one of the queues in $\Omega(A) \otimes \Omega(B)$ is empty. In fact, $(((s_{0,A}, \epsilon), (s_{0,B}, \epsilon)), (s_{0,A}, (s_{0,B}))) \in R$. Let $(((s_A, q_A), (s_B, q_B)), a, ((s'_A, q'_A), (s'_B, q'_B))) \in \Delta_{\Omega(A) \otimes \Omega(B)}$. We only consider transitions originating from $\Omega(A)$, the cases for transitions from $\Omega(B)$ are symmetric.

If $a \in T_A \cup \{\tau\}$, then $(s_A, a, s'_A) \in \Delta_A$ and $q_A = q'_A$, $(s_B, q_B) = (s'_B, q'_B)$. If $q_A = q_B = \epsilon$, then $(s_A, s_B) = (r_A, r_B)$ and thus $(((s'_A, q_A), (s_B, q_B)), (s'_A, r_B)) \in R$. If $q_A = \epsilon$ and $q_B \neq \epsilon$, then $(((s'_A, q_A), (s_B, q_B)), (r_A, r_B)) \in R$. But $q_A \neq \epsilon$ and $q_B = \epsilon$ is impossible, as then $s_A = r_A$ and hence A would not be input separated, since $q_A \in \Delta^*_A(r_A)$.

If $a \in T_{\Omega(A)} \setminus T_A$, then $a = m?$ for some $m \in I_A$, $(s_A, m, s'_A) \in \Delta_A$, $q_A = m \cdot q'_A$, $(s_B, q_B) = (s'_B, q'_B)$. Thus $s_A = r_A$, $(r_B, q_A, s_B) \in \hat{\Delta}^*_{B\xi}$, and $q_A \in \Delta_A(s_A)$. Moreover, $q'_A \in \Delta^*_A(s'_A)$. If $q'_A = \epsilon$, then $((r_A, r_B), m, (s'_A, s_B)) \in \Delta^*_{A\xi \otimes B\xi}$ and $(((s'_A, q'_A), (s_B, q_B)), s'_A, s_B) \in R$; if $q'_A \neq \epsilon$, $((r_A, r_B), m, (s'_A, r'_B)) \in \hat{\Delta}^*_{A\xi \otimes B\xi}$ with $(r'_B, q'_A, s_B) \in \hat{\Delta}_{B\xi}$ and hence $(((s'_A, q'_A), (s_B, q_B)), r'_A, r'_B) \in R$.

If $a \in O_{\Omega(A)}$, then $a = m!$ for some $m \in O_A$, $(s_A, m, s'_A) \in \Delta_A$, $q_A = q'_A$, $s_B = s'_B$, $q'_B = q_B \cdot m$. If $q_A = q_B = \epsilon$, then $(s_A, s_B) = (r_A, r_B)$. As $(r_A, r_B) \in \mathscr{R}(A \otimes B)$ and because A and B are compatible, there is a state $(s'_A, s'_B) \in S_{A \otimes B}$ with $((s_A, s_B), m, (s'_A, s'_B)) \in \Delta_{A \otimes B}$ and thus $(((s'_A, q_A), (s_B, q_B \cdot m)), (r_A, r_B)) \in R$. If $q_A = \epsilon$ and $q_B \neq \epsilon$, then $s_B = r_B$. We have $(r_A, q_B \cdot m, s'_A) \in \hat{\Delta}^*_{A\xi}$. In

order to show that $q_B \cdot m \in \Delta_B^*(r_B)$, let $r'_B \in S_B$ with $(r_B, q_B, r'_B) \in \Delta_B^*$. Then $((r_A, r_B), q_B, (s_A, r'_B)) \in \hat{\Delta}_{A\xi\otimes B\xi}$ and, in particular, $(s_A, r'_B) \in \mathscr{R}(A \otimes B)$ and hence, by the compatibility of A and B, we have $(r'_B, m, r''_B) \in \Delta_B$. Thus $(((s'_A, q_A), (s_B, q_B \cdot m)), (r_A, r_B)) \in R$. But $q_A \neq \epsilon$ and $q_B = \epsilon$ is impossible, as then $s_A = r_A$ and hence A would not be input separated, since $q_A \in \Delta_A^*(r_A)$. □

Theorem 1. *Let A and B be composable, input separated iotss without queue labels, and let $A \otimes B$ be closed. Let A and B be always eventually inputting. If A and B are compatible, then A and B are buffered compatible.*

Proof. Let A and B be compatible. Then $\Omega(A) \otimes \Omega(B)$ is half-duplex by Lem. 4. Consider an infinite weak trace λ of $\Omega(A) \otimes \Omega(B)$. As A and B are always eventually inputting, λ shows infinitely many labels of the form $m?$ with m an input label of A and infinitely many labels of the form $n?$ with n an input label of B. In each state of $\Omega(A) \otimes \Omega(B)$ with an outgoing transition with a label marked with ? the corresponding queue of A and B resp. is not empty, thus the queue of the other iots is empty. Hence each output of $\Omega(A)$ and $\Omega(B)$ is eventually answered by an input of A and B resp. and hence λ is buffered compatible. □

Example 8 (Compatiblity and buffered compatibility). The theorem is applicable to the iotss given by Fig. 3. The iotss are obviously input separated. They are compatible by Ex. 6, and they are always eventually inputting, since all of their weak infinite traces show infinitely many input labels. Therefore the iotss are indeed buffered compatible. Note that the input assumption for A and B is necessary. The iots on the left-hand side of Fig. 9 is not always eventually inputting and even though the iotss are synchronously compatible, they are not buffered compatible. The sender may proceed infinitely often with output actions while the receiver never dequeues resulting in an infinite weak trace that is not μ-buffered compatible.

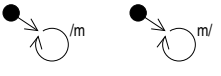


Fig. 9. Buffered incompatible I/O-transition systems

As witnessed by Ex. 7 (Fig. 7), the converse of Thm. 1, that buffered compatibility induces synchronous compatibility of the non-queued iotss, is not true in general.

6 Compositional Refinement of Connection-Safe Assemblies

When substituting a refined version of a component for another component in an assembly context, it should be ensured that relevant properties of the original assembly are preserved. In the following we introduce a notion of refinement of components based on alternating simulations of interface automata [4] and show that, at least for binary assemblies, component refinement is compositional and preserves connection-safety of assemblies.

6.1 Refinement of I/O-Transition Systems and Components

When an alternating simulation, as defined by de Alfaro and Henzinger [4], puts two states of an abstract and a concrete behaviour in relation, each input of the abstract behaviour must be accepted as an input of the concrete behaviour and, conversely, each output of a concrete behaviour must be an output of the abstract behaviour; these requirements correspond to clauses (1) and (2) in the definition below. We adopt this general idea, but formally extend the definition of alternating simulations in several points: Since we study connection-safety of assemblies, we do not allow a concrete behaviour to fall silent w.r.t. outputs, when the abstract behaviour showed some output; this condition is inspired by one part of stuck-freedom introduced by Rajamani and Rehof [14] and is represented in clause (5). As we are interested in the communication behaviour of assemblies, we do not abstract from internal actions and use an action τ for invisible behaviour; we require concrete internal actions to exist on the abstract level by clause (3). In contrast, concrete τ actions are optional on the abstract level as long as the iots-simulation relation is taken into account (cf. clause 4). Abstract internal and τ actions are treated by clause (6) and (7) like output actions in clause (5). Finally, and more technically, we remove the requirement of input determinism of interface automata, saying that for each input label there is at most one successor state. Instead, we introduce the weaker condition (8) below, that inputs of the concrete behaviour do not introduce more non-determinism than the corresponding inputs in the abstract behaviour.

Definition 12. *Let* $A = ((I_A, O_A, T_A), S_A, s_{0,A}, \Delta_A)$ *and* $C = ((I_C, O_C, T_C), S_C, s_{0,C}, \Delta_C)$ *be iotss such that* $I_A \subseteq I_C$, $O_C \subseteq O_A$ *and* $T_C \subseteq T_A$. *A relation* $R \subseteq S_A \times S_C$ *is an* alternating iots-simulation *for* A *and* C, *if for all* $(s_A, s_C) \in R$ *it holds that*

1. $\forall l \in I_A \,.\, \forall s'_A \in S_A \,.\, (s_A, l, s'_A) \in \Delta_A \implies (\exists s'_C \in S_C \,.\, (s_C, l, s'_C) \in \Delta_C \wedge (s'_A, s'_C) \in R)$,
2. $\forall l \in O_C \,.\, \forall s'_C \in S_C \,.\, (s_C, l, s'_C) \in \Delta_C \implies (\exists s'_A \in S_A \,.\, (s_A, l, s'_A) \in \hat{\Delta}_A \wedge (s'_A, s'_C) \in R)$,
3. $\forall l \in T_C \,.\, \forall s'_C \in S_C \,.\, (s_C, l, s'_C) \in \Delta_C \implies (\exists s'_A \in S_A \,.\, (s_A, l, s'_A) \in \hat{\Delta}_A \wedge (s'_A, s'_C) \in R)$,
4. $\forall s'_C \in S_C \,.\, (s_C, \tau, s'_C) \in \Delta_C \implies (\exists s'_A \in S_A \,.\, (s_A, \tau, s'_A) \in \hat{\Delta}_A \wedge (s'_A, s'_C) \in R)$,
5. $(\exists l' \in O_A \,.\, \exists s''_A \in S_A \,.\, (s_A, l', s''_A) \in \Delta_A) \implies (\exists l \in O_A \,.\, \exists s'_A \in S_A \,.\, \exists s'_C \in S_C \,.\, (s_A, l, s'_A) \in \Delta_A \wedge (s_C, l, s'_C) \in \Delta_C \wedge (s'_A, s'_C) \in R)$,
6. $(\exists a' \in T_A \,.\, \exists s''_A \in S_A \,.\, (s_A, a', s''_A) \in \Delta_A) \implies (\exists a \in T_A \,.\, \exists s'_A \in S_A \,.\, \exists s'_C \in S_C \,.\, (s_A, a, s'_A) \in \Delta_A \wedge (s_C, a, s'_C) \in \Delta_C \wedge (s'_A, s'_C) \in R)$,
7. $(\exists s''_A \in S_A \,.\, (s_A, \tau, s''_A) \in \Delta_A) \implies (\exists s'_A \in S_A \,.\, \exists s'_C \in S_C \,.\, (s_A, \tau, s'_A) \in \Delta_A \wedge (s_C, \tau, s'_C) \in \Delta_C \wedge (s'_A, s'_C) \in R)$.
8. $\forall l \in I_A \,.\, (\exists s''_A \in S_A \,.\, (s_A, l, s''_A) \in \Delta_A) \implies (\forall s'_C \in S_C \,.\, (s_C, l, s'_C) \in \Delta_C \implies (\exists s'_A \in S_A \,.\, (s_A, l, s'_A) \in \Delta_A \wedge (s'_A, s'_C) \in R))$,

The iots C *is a* refinement *of the iots* A, *written* $C \sqsubseteq A$, *if there exists an alternating iots-simulation* R *for* A *and* C *with* $(s_{0,A}, s_{0,C}) \in R$.

The concept of refinement can be immediately transferred to components by considering their observable behaviours:

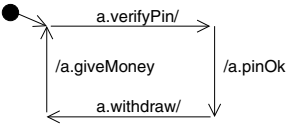


Fig. 10. Refined behaviour of Bank (cf. Fig. 3)

Definition 13. *A component C' is a* subtype *of a component C, written $C \geq C'$, if* $\mathrm{ports}(C) = \mathrm{ports}(C')$ *and* $obs(C) \sqsupseteq obs(C')$.

Example 9 (Refinement). In the context of our Bank–ATM example, the iots in Fig. 10 refines the observable behaviour of the component Bank given in Fig. 3, where the transition of outputting the message that the PIN is not correct has been removed. In particular, the new behaviour is more deterministic than the old one.

6.2 Refinement of Compatible I/O-Transition Systems

Let us first consider refinements of synchronously compatible iotss. In [4], de Alfaro and Henzinger proved that compatibility of input deterministic iots is preserved by their notion of alternating simulations. We extend this result to general iotss and our extended concept of refinement.

Theorem 2. *Let A, B and C be iotss. Let A, B and C, B be composable, and let $A \otimes B$ and $C \otimes B$ be closed. Let A and B be compatible. If $A \sqsupseteq C$, then $A \otimes B \sqsupseteq C \otimes B$ and C and B are compatible.*

Proof. Let $A = ((I_A, O_A, T_A), S_A, s_{0,A}, \Delta_A)$, and similarly for B and C. Let $A \otimes B = ((I_{AB}, O_{AB}, T_{AB}), S_{AB}, s_{0,AB}, \Delta_{AB})$ and $C \otimes B = ((I_{CB}, O_{CB}, T_{CB}), S_{CB}, s_{0,CB}, \Delta_{CB})$. Then $I_{AB} = O_{AB} = I_{CB} = O_{CB} = \emptyset$.

Let R_{AC} be an alternating iots-simulation for A and C with $(s_{0,A}, s_{0,C}) \in R$. Let

$$R = \{((s_A, s_B), (s_C, s_B)) \mid (s_A, s_C) \in R_{AC} \wedge (s_A, s_B) \in \mathscr{R}(A \otimes B)\} .$$

Then $((s_{0,A}, s_{0,B}), (s_{0,C}, s_{0,B})) \in R$. Let $((s_A, s_B), (s_C, s_B)) \in R$. We have to check clauses (1–8) for alternating iots-simulations for R. Since $I_{AB} = O_{AB} = O_{CB} = \emptyset$, clause (1), clause (2), clause (5) and clause (8) are satisfied vacuously. We only detail clause (3) for $l \in L(A) \bowtie L(B)$; clause (6) for these labels is analogous to (3) and the remaining cases merely transfer the alternating iots-simulation R_{AC} to R.

Let $l \in L(A) \bowtie L(B)$ and $((s_C, s_B), l, (s'_C, s'_B)) \in \Delta_{CB}$. Then $l \in O_A = I_B$ or $l \in I_A = O_B$ and $(s_C, l, s'_C) \in \Delta_C$, $(s_B, l, s'_B) \in \Delta_B$. If $l \in O_A = I_B$, by clause (2) for R_{AC}, there is an $s'_A \in S_A$ with $(s_A, l, s'_A) \in \hat{\Delta}_A$ and thus $((s_A, s_B), l, (s'_A, s'_B)) \in \hat{\Delta}_{AB}$ and also $((s'_A, s'_B), (s'_C, s'_B)) \in R$. If $l \in I_A = O_B$, then, since $(s_A, s_B) \in \mathscr{R}(A \otimes B)$ and A and B are compatible, there is an $s''_A \in S_A$ with $(s_A, l, s''_A) \in \Delta_A$. Thus there is an $s'_A \in S_A$ with $(s_A, l, s'_A) \in \Delta_A$ and $(s'_A, s'_C) \in R_{AC}$ by clause (8). Hence $((s_A, s_B), l, (s'_A, s'_B)) \in \hat{\Delta}_{AB}$ and also $((s'_A, s'_B), (s'_C, s'_B)) \in R$.

In order to show the compatibility of C and B, let $(s_C, s_B) \in \mathscr{R}(C \otimes B)$. Then there is an $(s_A, s_B) \in \mathscr{R}(A \otimes B)$ with $((s_A, s_B), (s_C, s_B)) \in R$, by induction using clause (3) for R. If $(s_C, l, s'_C) \in \Delta_C$ with $l \in O_C \cup T_C$, then there is an $s'_A \in S_A$ with $(s_A, l, s'_A) \in \Delta_A$ by clause (2) for R_{AC} and thus $(s_B, l, s'_B) \in \Delta_B$ for some $s'_B \in S_B$ by the compatibility of A and B. If $(s_B, l, s'_B) \in \Delta_B$ with $l \in O_B = I_A$, then there is an $s'_A \in S_A$ with $(s_A, l, s'_A) \in \Delta_A$ by the compatibility of A and B. Thus there is an $s'_C \in S_C$ with $(s_C, l, s'_C) \in \Delta_C$ by clause (1) for R_{AC}. □

From this theorem we immediately obtain the desired compositionality result for refinement in the case of synchronously compatible iots.

Corollary 1. *Let A, B, C, and D be iotss. Let A, B and C, B and C, D be composable, and let $A \otimes B$, $C \otimes B$, and $C \otimes D$ be closed. Let A and B be compatible. If $A \sqsupseteq C$ and $B \sqsupseteq D$, then $A \otimes B \sqsupseteq C \otimes D$ and C and D are compatible.* □

Example 10 (Refinement and compatibility). The iotss for the behaviours of the Bank–Atm application in Fig. 3 are compatible (modulo port relabelling). The iots in Fig. 10 is a refinement of the original behaviour of the Bank component. By application of Thm. 2 to a synchronous composition of the iotss, we may replace the original Bank behaviour by its refined version and obtain, first, that the composition is a refinement of the original composition and, second, that the refined iots is still (synchronously) compatible with the iots of the Atm behaviour.

For refinements in the context of buffered compatible asynchronous compositions the analogue of Thm. 2 holds. Here, clauses (5), (6) and (7) of Def. 12 for keeping at least one abstract output or internal label in the concrete behaviour are not only conceptually relevant, but also play a major technical role. For technical reasons, we restrict ourselves to a concrete input separated iots and we have to ensure that there are only infinite runs of the composition: An iots $A = (L, S, s_0, \Delta)$ is *deadlock free*, if for all $s \in \mathscr{R}(A)$ there is an $l \in \bigcup L$ and an $s' \in S$ with $(s, l, s') \in \hat{\Delta}$.

Theorem 3. *Let A, B and C be iotss without queue labels. Let A, B and C, B be composable, and let $A \otimes B$ and $C \otimes B$ be closed. Let A and B be buffered compatible, C input separated, and $\Omega(A) \otimes \Omega(B)$ deadlock-free. If $A \sqsupseteq C$, then $\Omega(A) \otimes \Omega(B) \sqsupseteq \Omega(C) \otimes \Omega(B)$ and C and B are buffered compatible.*

Proof. Let $A = (L_A, S_A, s_{0,A}, \Delta_A)$, and similarly for B, C, $\Omega(A)$, $\Omega(B)$, and $\Omega(C)$. Since A, B and C, B are composable, $\Omega(A), \Omega(B)$ and $\Omega(C), \Omega(B)$ are composable by Lem. 3. Let $\Omega(A) \otimes \Omega(B) = ((I_{AB}, O_{AB}, T_{AB}), S_{AB}, q_{0,AB}, \Delta_{AB})$, $\Omega(C) \otimes \Omega(B) = ((I_{CB}, O_{CB}, T_{CB}), S_{CB}, s_{0,CB}, \Delta_{CB})$. By the closedness of $A \otimes B$ and $C \otimes B$, we have $I_A = O_B = I_C$ and $O_A = I_B = O_C$, hence $I_{\Omega(A)} = O_{\Omega(B)} = I_{\Omega(C)}$ and $O_{\Omega(A)} = I_{\Omega(B)} = O_{\Omega(C)}$, and hence $I_{AB} = O_{AB} = I_{CB} = O_{CB} = \emptyset$. Furthermore $T_{AB} = T_{\Omega(A)} \cup L(\Omega(A)) \bowtie L(\Omega(B)) \cup T_{\Omega(B)}$ with $T_A \subseteq T_{\Omega(A)}$.

Let R_{AC} be an alternating simulation for A and C with $(s_{0,A}, s_{0,C}) \in R_{AC}$. Let

$$R = \{(((s_A, q), (s_B, q_B)), ((s_C, q), (s_B, q_B))) \mid (s_A, s_C) \in R_{AC} \wedge ((s_A, q), (s_B, q_B)) \in \mathscr{R}(\Omega(A) \otimes \Omega(B))\} .$$

Then $(((s_{0,A},\epsilon),(s_{0,B},\epsilon)),((s_{0,C},\epsilon),(s_{0,B},\epsilon))) \in R$. Let $(((s_A,q),(s_B,q_B)),((s_C,q),(s_B,q_B))) \in R$. We have to check the clauses (1–8) for alternating iots-simulations for R. Since $I_{AB} = O_{AB} = O_{CB} = \emptyset$, clause (1), clause (2), clause (5) and clause (8) are satisfied vacuously. We only detail clause (3) for $l \in (T_{\Omega(A)} \setminus T_A) \cup (L(\Omega(A)) \bowtie L(\Omega(B)))$; clause (6) for such labels is analogous to (3) and the remaining cases merely transfer the alternating iots-simulation R_{AC} to R.

If $l \in T_{\Omega(A)} \setminus T_A$ and $(((s_C,q),(s_B,q_B)),l,((s'_C,q'),(s'_B,q'_B))) \in \Delta_{CB}$. Then $l = m?$ for $m \in I_A$, $(s_C,m,s'_C) \in \Delta_C$, $q = m \cdot q'$, and $(s_B,q_B) = (s'_B,q'_B)$. Since $\Omega(A) \otimes \Omega(B)$ is deadlock-free and A and B are buffered compatible, there is an $a \in \bigcup L_A \cup \{\tau\}$ and an $s''_A \in S_A$ with $(s_A,a,s''_A) \in \Delta_A$; but $a \in O_A \cup T_A \cup \{\tau\}$ would contradict the input separation of C at s_C by clauses (5), (6) and (7) for R_{AC}, respectively. Thus $a \in I_A$ and $a = m$, since otherwise A and B would not be buffered compatible. In particular, there is an $s'_A \in S_A$ with $(s_A,m,s'_A) \in \Delta_A$ such that $(s'_A,s'_C) \in R_{AC}$ by clause (1) for R_{AC}. Thus $(((s_A,m \cdot q'),(s_B,q_B)),m?,((s'_A,q'),(s_B,q_B))) \in \hat{\Delta}_{AB}$ and also $(((s'_A,q'),(s_B,q_B)),((s'_C,q'),(s_B,q_B))) \in R$.

If $l \in L(\Omega(A)) \bowtie L(\Omega(B))$, then $l = m!$ with either $m \in O_A = I_B$ or $m \in I_A = O_B$. If $m \in O_A = I_B$, then $(s_C,m,s'_C) \in \Delta_C$, $q = q'$, $s_B = s'_B$, $q'_B = q_B \cdot m$. By clause (2) for R_{AC} there is an $s'_A \in S_A$ with $(s_A,m,s'_A) \in \hat{\Delta}_A$ and thus $(((s_A,q),(s_B,q_B)),m!,((s'_A,q),(s_B,q_B \cdot m))) \in \hat{\Delta}_{AB}$ and also $(((s'_A,q),(s_B,q_B \cdot m)),((s_C,q),(s_B,q_B \cdot m))) \in R$. If $m \in I_A = O_B$, then $(s_B,m,s'_B) \in \Delta_B$, $q' = q \cdot m$, $s_C = s'_C$, $q'_B = q_B$. Thus $(((s_A,q),(s_B,q_B)),m!,((s_A,q \cdot m),(s'_B,q_B))) \in \hat{\Delta}_{AB}$ and also $(((s_A,q \cdot m),(s'_B,q_B)),((s'_C,q \cdot m),(s'_B,q_B))) \in R$.

C and B are also buffered compatible: First, for each infinite run $p_0,l_0,p_1,l_1,\ldots$ of $\Omega(C) \otimes \Omega(B)$ we can inductively construct a simulating run $p'_0,l_0,p'_1,l_1\ldots$ of $\Omega(A) \otimes \Omega(B)$ such that $(p_k,p'_k) \in R$ for all $k \in \mathbb{N}$: if $(p_k,p'_k) \in R$, then we have $(p_k,l_{k+1},p_{k+1}) \in \hat{\Delta}_{CB}$ and thus there is a $p'_{k+1} \in S_{AB}$ with $(p'_k,l_{k+1},p'_{k+1}) \in \hat{\Delta}_{AC}$ and $(p_{k+1},p'_{k+1}) \in R_{AC}$ by clause (3) for R. Thus, if there would be an infinite weak trace of $\Omega(C) \otimes \Omega(B)$ which is not buffered compatible, there would be an infinite weak trace of $\Omega(A) \otimes \Omega(B)$ which is not buffered compatible, contradicting the buffered compatibility of A and B. □

From Thm. 3 we immediately obtain the desired compositionality result for the refinement of buffered compatible iotss.

Corollary 2. *Let A, B, C, and D be iotss without queue labels. Let A, B and C, B and C, D be composable, and let $A \otimes B$, $C \otimes B$, and $C \otimes D$ be closed. Let C and D be input separated. Let A and B be buffered compatible, and $\Omega(A \otimes B)$ and $\Omega(C \otimes B)$ deadlock-free. If $A \sqsupseteq C$ and $B \sqsupseteq D$, then $\Omega(A) \otimes \Omega(B) \sqsupseteq \Omega(C) \otimes \Omega(D)$ and C and D are buffered compatible.* □

Example 11 (Refinement and buffered compatibility). As an example for the application of Thm. 3 consider the observable behaviours in Fig. 3. Assume again, that the Bank behaviour is refined by an iots as given by Fig. 10. In order to apply the theorem we need to make sure that (1) the iotss in Fig. 3 are buffered compatible, (2) the product of the corresponding buffered iotss is deadlock-free and (3) the refined behaviour is input separated. (1) follows from Ex. 8, (2) is derived from the product in Fig. 4 and (3) is obvious from Fig. 10.

6.3 Substituting Components in Connection-Safe Assemblies

Having shown that refinement preserves (synchronous) compatibility and buffered compatibility, we can finally apply this result to the refinement of components in connection-safe assemblies and show that, under some mild restrictions, connection-safety is preserved when substituting components by refined components. As a technical prerequisite we note that refinement is a pre-congruence w.r.t. relabellings which preserve the kinds of labels:

Lemma 5. *Let A and C be iotss with $L(A) = (I_A, O_A, T_A)$ and $L(C) = (I_C, O_C, T_C)$ and $I_A \subseteq I_C$, $O_C \subseteq O_A$, and $T_A = T_C$. Let $\rho : (I_C, O_A, T_A) \to (I, O, T)$ be a relabelling with $\rho(l_i) \in I$, $\rho(l_o) \in O$, $\rho(l_t) \in T$ for $l_i \in I_C$, $l_o \in O_A$, and $l_t \in T_A$. If $A \sqsupseteq C$, then $A\rho \sqsupseteq C\rho$.* □

Theorem 4. *Let $\langle c : C, d : D; k : K\rangle$ be an assembly with $\mathrm{ports}(C) = \{p : P\}$ and $\mathrm{ports}(D) = \{q : Q\}$ and $K = (c.p : P, d.q : Q)$. Let C' and D' be components. If k is asynchronous, let $obs(C')$ and $obs(D')$ be input separated and let $\mathrm{beh}(\langle c : C, d : D; k : K\rangle)$ and $\mathrm{beh}(\langle c : C', d : D; k : K\rangle$ be deadlock-free. If $C \geq C'$, $D \geq D'$ and $\langle c : C, d : D; k : K\rangle$ is connection-safe, then $\langle c : C', d : D; k : K\rangle$ is connection-safe.*

Proof. Let $a = \langle c : C, d : D; k : K\rangle$ and $a' = \langle c : C', d : D'; k : K\rangle$. Then $\mathrm{beh}(a) = obs(c : C)\alpha\sigma \otimes obs(d : D)\alpha\sigma \otimes buf(k : K)$ and $\mathrm{beh}(a') = obs(c : C')\alpha\sigma \otimes obs(d : D')\alpha\sigma \otimes buf(k : K)$. Moreover, $\mathrm{beh}(a)$ and $\mathrm{beh}(a')$ are closed. Let a be connection-safe and $C \geq C'$, $D \geq D'$.

If k is synchronous, then $buf(k : K) = \mathbf{1}$ and α is the identity relabelling. The connection-safety of a thus amounts to the compatibility of $obs(c : C)\sigma$ and $obs(d : D)\sigma$; and a' is connection-safe, if, and only if $obs(c : C')$ and $obs(d : D')$ are compatible. From $C \geq C'$ and $D \geq D'$, we have $obs(C) \sqsupseteq obs(C')$ and $obs(D) \sqsupseteq obs(D')$ and thus $obs(c : C)\sigma \sqsupseteq obs(c : C')\sigma$ and $obs(d : D)\sigma \sqsupseteq obs(d : D')\sigma$ by Lem. 5. From Cor. 1, it follows that $obs(c' : C)$ and $obs(d : D')$ are compatible.

If k is asynchronous, then σ is the identity relabelling and by Prop. 1 we have $\mathrm{beh}(a) = com_{\{c.p:P,d.q:Q\}}(c : C)\kappa \otimes com_{\{c.p:P,d.q:Q\}}(d : D)\kappa$ which is the same as $\Omega_{X_p,Y_p}(obs(c : C))\kappa \otimes \Omega_{X_q,Y_q}(obs(d : D))\kappa$ by Def. 8 with $X_p = \{c.p.m \mid m \in \mathrm{msg}(\mathrm{prv}(P))\}$, and analogously for Y_p, X_q, Y_q. Now $\Omega_{X_p,Y_p}(obs(c : C))\kappa = \Omega(obs(c : C)\kappa)$ and similarly for $d : D$, as C and D have only a single port each and κ is a match relabelling which does not introduce queue labels. Thus the connection-safety of a amounts to the buffered compatibility of $obs(c : C)\kappa)$ and $obs(d : D)\kappa$; and a' is connection-safe, if, and only if $obs(c : C')\kappa$ and $obs(d : D')\kappa)$ are buffered compatible. But $C \geq C'$ and $D \geq D'$ induce $obs(c : C)\kappa \sqsupseteq obs(c : C')\kappa$ and $obs(d : D)\kappa \sqsupseteq obs(d : D')\kappa$ by Lem. 5 and leave $obs(c : C')\kappa$ and $obs(d : D')\kappa$ input separated; hence $obs(c : C')\kappa$ and $obs(d : D')\kappa$ are buffered compatible by Cor. 2. □

Example 12. [Connection-safe assemblies] The behaviours of the Bank–ATM application discussed so far are readily applicable to illustrate Thm. 4. For the implication in Fig. 11 to hold, we need to meet two assumptions in case of an asynchronous connector: first the behaviour of the subtype Bank' must be input separated, i.e. the component

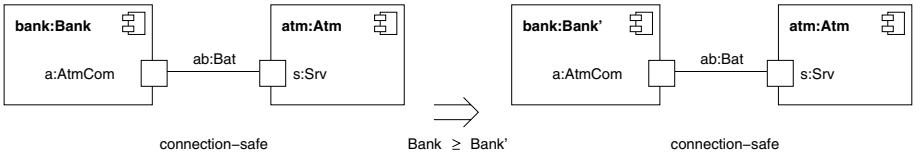


Fig. 11. Subtype substitution preserves connection safety

should behave like a single-threaded system and, second, the original assembly must be deadlock-free. Then connection-safety is preserved when replacing component Bank by its subtype Bank'.

Let Bank and Atm be components with observable behaviours as in Fig. 3. Let Bank' be a component with $\mathrm{ports}(\mathsf{Bank}) = \mathrm{ports}(\mathsf{Bank'})$ and an observable behaviour as in Fig. 10. Then,

- $\langle \mathsf{bank} : \mathsf{Bank}, \mathsf{atm} : \mathsf{Atm}; \mathsf{ab} : \mathsf{Bat} \rangle$ is connection-safe due to Ex. 6 if the connector is synchronous and due to Ex. 8 in the asynchronous case,
- $\mathsf{Bank} \geq \mathsf{Bank'}$ holds due to Ex. 9
- the observable behaviour of Bank' is obviously input separated,
- $\langle \mathsf{bank} : \mathsf{Bank}, \mathsf{atm} : \mathsf{Atm}; \mathsf{ab} : \mathsf{Bat} \rangle$ is deadlock-free, since the product of the corresponding buffered iotss in Fig. 4 is deadlock-free.

Hence $\langle \mathsf{bank} : \mathsf{Bank'}, \mathsf{atm} : \mathsf{Atm}; \mathsf{ab} : \mathsf{Bat} \rangle$ is connection-safe by Thm. 4. □

7 Conclusions

We have presented a component model which supports synchronous and asynchronous communication. For the formal foundation of behaviours we have used I/O-transition systems. The main focus has been on the study of communication behaviours between components in component assemblies. As a crucial desirable property we have required connection-safety of component assemblies which relies on compatibility conditions for iotss with synchronous and asynchronous communication. We have shown that synchronous compatibility is a sufficient criterion for buffered compatibility in asynchronous communications if both communication partners show observable behaviours which are input separated and always eventually inputting. Moreover, we have defined a refinement relation which is compositional w.r.t. synchronous and asynchronous connections of components and which preserves connection-safety.

Our compositionality results are proved for closed systems with only two connected components which already involves a lot of technical efforts due to the formal treatment of asynchronous communication with buffering behaviours. We believe that these results provide a solid basis for an extension of our theorems to closed assemblies with an arbitrary number of components. For the case of open systems further investigation incorporating assumptions on the environment as considered e.g. in [18] is necessary.

References

1. Lau, K.K., Wang, Z.: Software Component Models. IEEE Trans. Softw. Eng. 33(10), 709–724 (2007)
2. Rausch, A., Reussner, R., Mirandola, R., Plášil, F. (eds.): The Common Component Modeling Example. LNCS, vol. 5153. Springer, Heidelberg (2008)
3. Hennicker, R., Janisch, S., Knapp, A.: On the Observable Behaviour of Composite Components. In: Proc. 5^{th} Int. Wsh. Formal Aspects of Component Software, FACS 2008 (2008), http://www.pst.ifi.lmu.de/veroeffentlichungen/hennicker-et-al:facs:2008.pdf
4. de Alfaro, L., Henzinger, T.A.: Interface-based Design. In: Broy, M., Grünbauer, J., Harel, D., Hoare, C.A.R. (eds.) Engineering Theories of Software-intensive Systems. NATO Science Series: Mathematics, Physics, and Chemistry, vol. 195, pp. 83–104. Springer, Heidelberg (2005)
5. Bernardo, M., Ciancarini, P., Donatiello, L.: Architecting Families of Software Systems with Process Algebras. ACM Trans. Softw. Eng. Meth. 11(4), 386–426 (2002)
6. Brim, L., Černá, I., Vařeková, P., Zimmerova, B.: Component-Interaction Automata as a Verification-Oriented Component-Based System Specification. SIGSOFT Softw. Eng. Notes 31(2), 1–8 (2006)
7. Brand, D., Zafiropulo, P.: On Communicating Finite-State Machines. J. ACM 30(2), 323–342 (1983)
8. Jéron, T., Jard, C.: Testing for Unboundedness of Fifo Channels. Theo. Comp. Sci. 113, 93–117 (1993)
9. Maréchal, O., Poizat, P., Royer, J.C.: Checking Asynchronously Communicating Components Using Symbolic Transition Systems. In: Meersman, R., Tari, Z. (eds.) OTM 2004. LNCS, vol. 3291, pp. 1502–1519. Springer, Heidelberg (2004)
10. Poizat, P., Royer, J.C.: A Formal Architectural Description Language based on Symbolic Transition Systems and Temporal Logic. J. Univ. Comp. Sci. 12(12), 1741–1782 (2006)
11. Hacklinger, F.: JAVA/A – Taking Components into Java. In: Proc. 13^{th} ISCA Int. Conf. Intelligent and Adaptive Systems and Software Engineering (IASSE 2004), ISCA, pp. 163–169 (2004)
12. Ahumada, S., Apvrille, L., Barros, T., Cansado, A., Madelaine, E., Salageanu, E.: Specifying Fractal and GCM Components with UML. In: 26^{th} Int. Conf. Chilean Computer Science Society (SCCC 2007), pp. 53–62. IEEE, Los Alamitos (2007)
13. Barros, T., Boulifa, R., Madelaine, E.: Parameterized Models for Distributed Java Objects. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 43–60. Springer, Heidelberg (2004)
14. Rajamani, S.K., Rehof, J.: Conformance Checking for Models of Asynchronous Message Passing Software. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 166–179. Springer, Heidelberg (2002)
15. Cécé, G., Finkel, A.: Verification of Programs with Half-Duplex Communication. Inform. Comp. 202(2), 166–190 (2005)
16. Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)
17. Gouda, M.G., Manning, E.G., Yu, Y.T.: On the Progress of Communications between Two Finite State Machines. Inform. Contr. 63(3), 200–216 (1984)
18. Larsen, K.G., Nyman, U., Wasowski, A.: Interface Input/Output Automata. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 82–97. Springer, Heidelberg (2006)

Experiences in Model Driven Verification of Behavior with UML*

Fabrice Kordon and Yann Thierry-Mieg

LIP6, Université Pierre et Marie Curie,
4 place Jussieu, 75005 Paris, France
{Fabrice.Kordon,yann.thierry-mieg}@lip6.fr

Abstract. Model Driven Development (MDD) focuses on the intensive use of models during software development. In this paradigm, models are the central development artifact: transformations are used to derive executable programs, or tests for a given platform. This makes building quality models a cost-effective approach, as the models can be reused for many analysis or generation goals, and not just document a design. However, high quality models are needed for the approach to be successful. Hence the goal of performing analysis of high-level behavioral specifications such as UML, to enhance their quality and detect defects or ensure desired behavior.

High-level specifications provide many facilities to handle large specifications (such as hierarchical structuring mechanisms) and provide sophisticated features to handle programming language's rich semantics. However, the price of these features is that these specifications are difficult to analyse, the semantics are not necessarily formally defined, and the complexity of the language features usually limits analysis to manual inspection, or in the best cases simulation.

On the other hand, formal specifications have been developed specifically with analysis purposes in mind. In particular, model checking is an automatic approach suitable to analyse formally defined behaviors. However, formal specifications languages such as CSP, PROMELA, Petri nets, etc. have a steep learning curve, and are not cost effective since they are not directly linked to code.

In this paper, we explore an approach to integrate formal methods with high-level notations, by translating high-level specifications to formal ones to enable analysis. We are thus bringing Model Driven Engineering to Verification Driven Engineering. We show how this approach was put in practice with UML within the context of the ModelPlex project.

1 Introduction

Industry has always faced a major challenge in the design and implementation of complex systems: how to ensure that they behave appropriately. To do so,

* This work has been partially supported by the ModelPlex European integrated project FP6-IP 034081 (Modeling Solutions for Complex Systems).

C. Choppy and O. Sokolsky (Eds.): Monterey Workshop 2008, LNCS 6028, pp. 181–200, 2010.

the emerging consensus emphasizes the need for models. We use "models" in the broad sense, meaning tool manipulable descriptions of software design and artifacts. Although simulation and testing are the most common tools used to improve quality assurance, they only increase user's confidence in a piece of software, since by nature they are not exhaustive techniques.

Furthermore, in distributed systems that are becoming the norm, the inherent asynchronous behavior produces non-determinism that testing may have trouble detecting correctly. Only formal methods are capable of proving properties for such systems.

Current practice of software development strongly gravitate around Model Driven Engineering (MDE) that provides guidelines on how to elaborate models at each step of the development process. However, with the notable exception of hardware design, formal specifications are not widely accepted because they are difficult to operate:

- Their application to real-size systems most often requires heavy use of abstractions and tricky encodings of the specification,
- They must be operated using an appropriate formal notation and the related tools. Each one has its strengths and weaknesses, and all are relatively complex to acquire for an engineer,
- The models produced for verification of a design are usually not reusable for other goals such as code generation. Thus a rupture exists between the models and reality, and the investment in the models is considered too expensive for the quality assurance results provided.

This process usually require highly skilled engineers in both their application domain and various formal methods. These people are difficult to find. Thus, adoption by industry remains limited to niche applications, partly because the formalisms used for model checking purposes are considered too difficult to be used by average developers.

In contrast, Model-Driven Engineering (MDE) techniques are gaining attention since machine-readable specifications (models) are more precise, less error prone, can be processed by automated tools, and can allow by code generation to be less dependent on fast evolving technologies [19].

So, the use of formal methods suffer from methodological and technical issues and decrease the benefits of MDE. In [24], we proposed to define *Verification Driven Engineering* (VDE) as an an addition to the use of models all over the software development cycle. Since there is no "silver bullet", the idea is to provide enough information to help picking up the correct technique and tools. By providing model transformations and self configuration mechanisms to choose the adapted formal verification techniques and tools, we can increase their use by industry.

Models and related notations are crucial. When a community uses a notation, and links from that notation to formal methods can be operated, then it is possible to exploit tools and help designers to build safer systems while requiring limited knowledge of underlying techniques.

The objective of this paper is to show how VDE can be operated to provide such help to engineers in an automated way, to answer simple questions such as deadlock detection, bounds of resources, etc. To do so, we propose a general schema for relating high-level notations to formal methods (section 2). Then, this schema is instantiated for UML (section 3) and we present an implementation on some useful properties (section 4) before a conclusion.

2 Connecting High-Level Notations with Formal Methods

It is now widely accepted that models should be throughout the design and implementation procedure. However, our specific goal here is to make the best possible usage of these models, for example to check if expected behavioral properties are verified or not.

To perform verification through model-checking, a process like the one described in figure 1 is required. First, it is important to have specifications and associated properties (for example, invariants modeling safety properties). These can be expressed using standard high-level notations of an application domain such as UML.

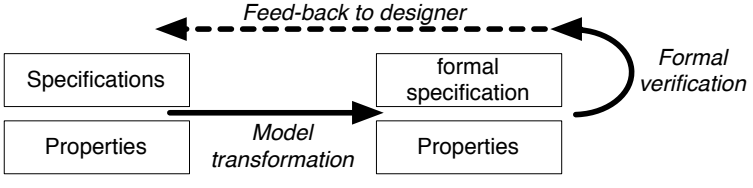


Fig. 1. Development process involving models and verification

Most high-level specifications are not directly suitable for the computation of properties. Therefore, they must be transformed into formal specifications on which properties can be verified. When the procedure succeeds a positive answer or an error diagnostic is provided. The verification procedure however may fail due to time or memory constraints. This error diagnostic feedback allows system architects to improve their design.

In an iterative development cycle, model validation is introduced in every iteration. It thus is applied on specifications that are successively refined. As suggested in [24], the MDE (Model Driven Engineering) process come to a VDE (Verification Driven Engineering) process. Figure 2 shows the overall development process that is helicoidal. Each step corresponds to a version of the system specifications that are verified and then, potentially corrected or enriched.

We first briefly investigate several types of verification techniques and then identify the main issues to be solved when using formal methods in a VDE approach. Section 3 shows how such an approach can be instantiated for UML.

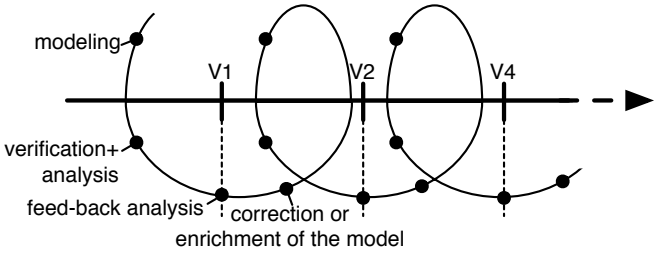


Fig. 2. Helicoidal development life cycle

UML is by far the most popular and tool-supported language used for industrial specifications [14]. For the formal verification community, this offers the possibility of defining model checking based verification as one of many services on industrial UML models. It thus helps adoption of these formal techniques into mainstream software development methods.

2.1 Choosing a Class of Verification Techniques

There are several classes of formal verification techniques that allow one to reason on a model based on formal grounds. Through different theories (sets, automata, stochastic, etc.), there is a large panel of methods. Let us list two of them.

Algebraic Approaches and Axiomatic Logic. such as Z [23] or B [1] allow to describe a system using axioms and then, prove properties of this specification as a theorem to be demonstrated from these axioms. For example, these methods allow one to check for the consistency of interfaces through a complete type checking mechanism, or even to go further and prove theorems (lemmas, invariants) on a system.

These are of particular interest because the proof is parametric and abstract ; for instance a property can hold for a number of entities taken in the natural range. However, theorem provers that help elaborate the proof are difficult to use and still require highly skilled and experienced engineers.

Model Checking. [10] is an active research domain [11]. It consists in the exhaustive investigation of a system state space. A designer expresses a property to be tested on a model, using a temporal logic formula expressing (un)desired behaviors of the system. This formula is compared with all the paths in the system's state space. If there is a path that does not satisfy the property, then the property does not hold and the returned path (or execution trace) exhibits a counter-example for the property.

The main advantage of this technique is that it is fully automated given a system and a property. However, results obtained are rarely parametric, they answer for a particular set of resources (e.g. N threads). Besides, model-checking

is limited by the combinatorial explosion and can mainly address finite systems. However, many efficient techniques exist such as symbolic decision diagram (BDD) [5], partial order and abstraction methods [12], or symmetry based techniques [7]. They allow to scale up to more complex systems. Some recent studies also investigate model checking of infinite-state systems [25]. Other extensions allow the verification of time-related or probabilistic properties on a model.

The complexity of current software systems and their parallel, distributed and heterogeneous nature raises many challenges when trying to ensure their correctness. In this respect, *model checking* [22] is a formal verification technique which promises to automatically check whether a system satisfies some stated properties. Thus, it seems that model checking is currently a better solution for building push-button tools implementing VDE.

2.2 Raised Issues

However, several important issues must be solved in order to operate such an approach:

- specification issues
- transformation issues
- complexity issues

Specification Issues. To be suitable for verification purpose, the specification must contain precise information about the behavior of its components. If the description is structured, an appropriate way of combining behaviors of components must be provided.

For instance, in a notation like UML, information is consistent in diagrams but lacks semantics when diagrams are connected. For example, state-machines describe the behavior of a *Classifier* (*Class* or *Component*). But how this behavior is connected to a description based on sequence diagrams is rather unclear. Therefore, if verification can be performed on isolated diagrams like in [32] for state-machines or in [13] for sequence diagrams, it is difficult to ensure consistency of the entire system specification.

To overcome this first problem, profiling can be considered. The missing semantic information can be stated by means of annotations that are exploited to fix identified "semantic variation points" and other places where the semantic is imprecisely defined.

Another important point concerns the specification of properties. Invariant specification is currently used in industry, for example to specify the profile of unexpected events in the system. OCL is the standard notation to express invariants in UML specifications. However, temporal logic is more difficult to use and is not fully standardized.

To overcome this second problem, sequence diagrams can be used as a more friendly way of specifying temporal logic formulae. These diagrams provide a way to help designers to describe expected causalities between events (since

sequence diagrams express a partial order over events). These descriptions of expected behavior can be used for verification.

Let us note that a notation like AADL [29] solves these problems by providing a unified way to relate component to annotations[1]. AADL components describe a piece of the system while annotations define either component features or component properties. Thus, both the system, its behavior (in an annotation) and its expected properties are defined together. This is helpful when transforming the full specification of a system into a formal notation.

Transformation Issues. High-level concepts must be mapped to formal specifications. When elaborating the transformation rules, it is important to consider:

- The granularity of atomic events: it must be tuned appropriately to limit complexity and while remaining precise in the specification.
- The traceability of transformation to provide comprehensive feed-back to engineers in terms of the high-level specification.
- The consistency between the transformation of the specification and its associated properties, particularly if different modeling notations are used.

Since no formal verification technique fits all the needs it is necessary to choose the most appropriate one. Therefore, several transformations from the high-level notation to various formal models must be considered. Each transformation is dedicated to the verification of a given property using a given technique.

Complexity Issues. Combinatorial explosion in the verification process is another reason to consider several transformation. It is important to select, for a given formal notation, the most efficient verification technique. Such a technique depends on the formal notation but also on the property to be verified. Therefore, there can be several transformation from the high-level notation to a given formal notation, each one being optimizing some configuration.

As an example, let us consider deadlock detection in Petri nets. It can be achieved by means of model checking but also trough structural analysis for certain types of Petri Nets like in [2,3]. The structural approach usually scales much better, though it is less general than full model-checking. Customizing a transformation for deadlock detection is thus of interest.

2.3 An Approach to Enable VDE

The elaboration of a VDE approach requires a methodology: one must cope with the issues and elaborate its transformation and verification tools. The approach we propose has four steps:

1. Selection of the semantic scope to be handled in the high-level specification. This is necessary to bound the transformations rules. It is possible to extend the scope when a first set of transformation rules are validated.

[1] In AADL, annotations are called "properties".

2. Selection of the appropriate formal notation according to the type of property to be verified. It is also necessary to identify the technique to be used for verification.
3. Identification of the abstraction level to be considered in the high-level specification. This corresponds to the selection of the elements to be considered in the high-level notation for the transformation to be elaborated.
4. Mapping of the selected elements to the formal notation. It is important to identify the elements that correspond to the "glue" in the system such as hierarchical composition or communication mechanisms. The transformation rules handling these elements are of particular importance because they orchestrate others.

Through these phases, it is important to preserve traceability information so that results can be expressed in the original notation.

3 From UML to Formal Methods

Model-Driven Engineering (MDE) development methods are gaining increasing attention from industry. In MDE, the model is the primary artifact and serves several goals among which code generation, requirements traceability, and model-based testing. MDE thus enables cost-effective building of models vs. direct coding of an application. In this context, model-based formal verification of behavioral consistency is desirable as it helps improve model quality.

In this section we present a translation based approach using (parts of) UML [28] as source and Instantiable Petri Nets (IPN) as target, to enable formal verification. IPN is a hierarchical formalism defined expressly for this purpose; it is not meant to be used as a front-end modeling language but rather as a powerful back-end verification formalism. The semantics and concepts are thus kept simple, they are described in detail in [30]. We show that IPN are adequate to support the introduction of model checking in an MDE process. The approach is implemented in a prototype tool called BCC: Behavioral Consistency Checker.

3.1 Verification of UML

UML is a standard defined using a meta-modeling approach to establish the concepts of the various diagrams, and plain English to describe dynamic aspects of the specification. UML2 has introduced an important refactoring of the description of actions and behaviors, which allows to describe the behavioral diagrams (activity, state machine and interaction diagrams) using a common base. The semantics of a UML2 model is thus more precise than in UML 1.x. But so-called semantic variation points are deliberately left in the standard, to help UML fit all possible application fields of software engineering. The semantics of UML thus remains imprecise and subject to interpretation.

The idea of providing formal semantics to (at least parts) of the UML by translation to a more formal description is widely used [4]; it allows to use the UML as

a (relatively) user friendly graphical environment and exploit the existing formal verification techniques and tools without redeveloping them specifically for the UML. It also is compatible with the ideas of Model-Driven Engineering (MDE) in which the model is the central artifact, and translations from the model to other artifacts (code, tests...) with specific goals in mind is the norm.

One of the challenges when defining such a translation is to handle the composition of parts of a behavior, defined using different types of diagrams. This touches on the problem of consistency among UML diagrams of a given system: although structural consistency rules are defined (using OCL) and can be checked mostly syntactically, the multiple viewpoints offered by the UML induce the possibility of defining incompatible behaviors quite easily.

Given the wealth of work on translation of UML models for analysis purposes, we focus here on a few propositions which are the most similar to this work, particularly those targeting dialects of Petri nets in the translation.

Merseguer et al. have done an important work ([6] gives an overview) on translation of the UML to labeled generalized stochastic Petri nets (LGSPN) for performance analysis purposes. Tool support is provided through the ArgoSPE tool set and the model checker GreatSPN. This work is perhaps the closest to ours because the semantics used to compose the LGSPN is based on synchronizations, like in our IPN. However, that work is centered on performance evaluation rather than consistency checking. Moreover LGSPN do not allow hierarchical compositions.

The work of Eshuis et. al [17] also gives a formal semantics to activity diagrams through translation to workflow nets, and more recently to NuSMV. However they concentrate mainly on a subset of activity diagrams compatible with UML 1.5, and do not handle hierarchy of the description.

Shatz et al. have also done a large body of work [21,31] on formalization of UML semantics, mostly centered on state charts, that uses Petri nets as translation target. The focus of that work is in correctly capturing the full run to completion semantics of UML state machines. It thus does not address the problem of inter diagram consistency checking.

Engels et al. [16] have elaborated a formalization framework based on CSP as semantic domain. This work is mainly focused on protocol state charts of UML 1.4. It does not handle hierarchy as to enable scalability in the specification.

Overall the main originality of this work is in the composition mechanisms used and the fact we natively support hierarchy instead of requiring a flattening of the representation, which leads to scalability issues. We also present an implementation in section 4.

3.2 Instantiable Petri Nets

In this work we have used Instantiable Petri Nets (IPN) as a target in the transformation. This formalism is presented in detail in [30], we give an informal presentation here sufficient to explain the subsequent transformation.

IPN explicitly support the concepts of *type and instance* that allow to adequately match the structure of high-level models designed with UML. They also

allow to reuse parts of a model in various scenarios, and offer good scale up properties to handle large and complex specifications. Finally, since instances of a type share a common definition, they allow to capture internal regularity or symmetries of a system, which can be exploited for model checking.

IPN explicitly support a compositional definition based on the notion of *synchronizations*. A wide corpus of theoretical [18] and empirical [8] results (introduced by process algebra such as CSP [26], up to applicability to probabilistic systems [15]) show that this mode of composition is favorable to better compositional verification algorithms.

Their definition is split in two parts.

First, we define Elementary Petri Nets, which are essentially standard Petri nets in which transitions bear a *visibility* that may be private or public. A public transition is part of the *interface* of the net; it can only be fired if it is sollicited through an external synchronization. Transitions private to a net can be fired according to usual Petri net semantics.

We then define a Composite type, to hold instances of elementary nets, or, recursively, instances of a composite type. A Composite may contain synchronizations, again labeled with a visibility. A synchronization forces synchronous firing of its *parts*, that is transitions belonging to the interface of the contained instances.

This hierarchical formalism uses a semantic largely inspired by process calculi such as CSP. It also borrows from component based formalisms (such as Corba component model (CCM), or Fractal) the notions of components defined as hierarchical composition of simpler bricks.

Examples of IPN will be provided in the next sections, a formal definition of IPN can be found in [30].

3.3 How to Translate

The issues relating to transformation of UML to a formal notation are explained here through an example.

We use in this paper UML activity diagrams. A similar approach can be applied to other UML diagrams that represent a *Behavior*. The essential characteristic of a behavior is that it begins with an occurrence of a start event and ends with a termination event occurrence (UML Superstructure, section 13, p.419 [28]).

The transformation is based on a set of patterns of transformation. The principle consists in building one IPN type per diagram of the UML specification. These types will then be instantiated and assembled according to various verification scenarios. Hence for each activity diagram we build an elementary IPN type.

We first apply the translation rules for the various types of *nodes* of the UML activity diagram, using the patterns described in figure 3.

Places: Each node gives rise to one or more places and zero or more public transitions. The most complex case is the *callBehavior* pattern where the place b represents a state where we are waiting for the called behavior to complete. We keep track during the application of this transformation of the *in* and *out* places

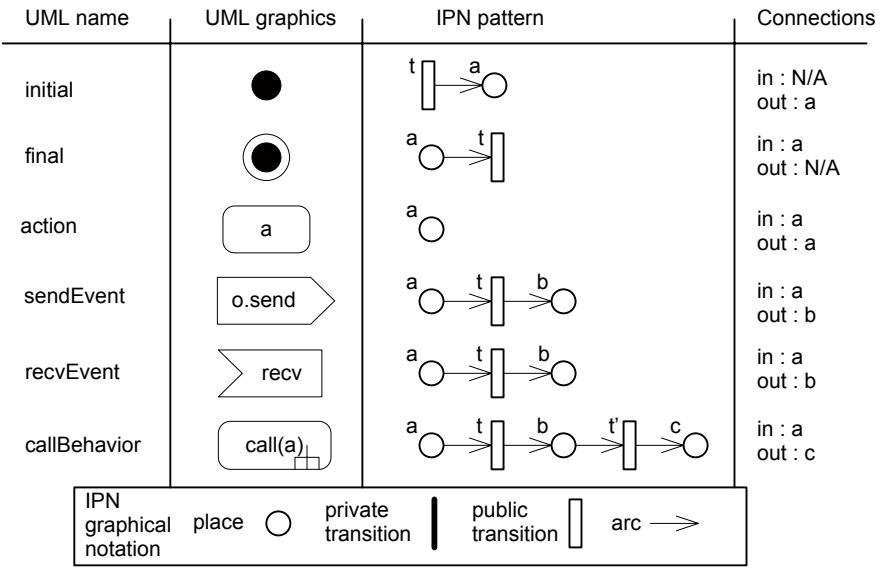


Fig. 3. Translation rules for nodes of UML2 activity diagrams. Patterns are expressed here in a simplified form: each generated public transition is also associated to sufficient information to enable appropriate connections in the linking phase (see section 3.4).

generated for each UML object. After this first pass all places of the resulting IPN have been produced; each control flow has a source (the "out" place of the source activity) or a destination (the "in" place of the target activity) or both. An additional case not represented on this figure (but used in the example see fig. 6) arises for control flows that link two control nodes (e.g. merge to fork): we produce an additional place for these edges that acts as both as source and destination in the translation.

Interface: The public transitions produced are meant to be synchronized with other diagrams: the transition labeled t of the *initial* pattern is meant to be synchronized with the transition t of the *callBehavior* pattern. Similarly, the transition t of the *final* pattern is meant to be synchronized with the transition t' of the *callBehavior* pattern. Transition t of the *sendEvent* pattern is meant to be synchronized with the appropriate corresponding transition t of the *recvEvent* pattern of the receiving object.

States: We additionally define two labeled states for each diagram, *active* which assigns one token to the place corresponding to the UML initial node, and *passive* (the default) in which all places are initially empty.

Transitions: We then translate the control flows and control nodes of the UML activity diagram, using the patterns described in figure 4. In these patterns, the activities noted a, b and c are just placeholders for the actual nodes that

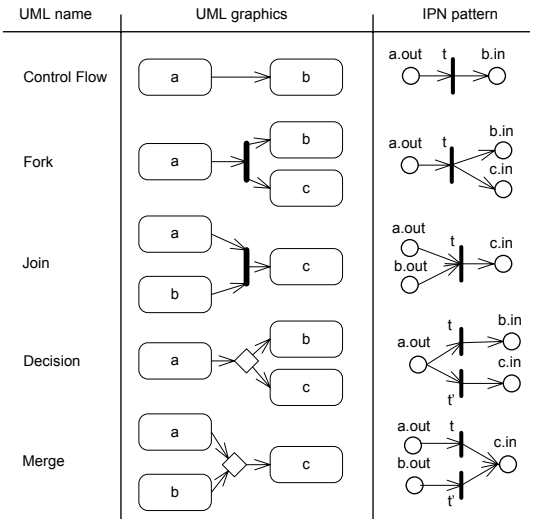


Fig. 4. Translation rules for edges of UML2 activity diagrams

the edges connect to: the translation pattern is centered on a control node, and queries the control flows that link to it to obtain the appropriate source or target place that was defined in the first translation step.

These patterns are mostly straightforward Petri net interpretations of UML semantics, so we hope a reader familiar with both UML and Petri nets will understand these translation patterns without further ado.

All the transitions produced in this phase have private visibility: interaction with other diagrams happens within nodes in activity diagrams. This is to be contrasted with the reactive semantics of UML state-machines, in which edges are the main point of interaction with other diagrams.

Application to an Example. Figure 5 presents a small example that describes an order and shipping process. It contains many of the UML features our translation supports. The action HandlePayment of the "Order" activity is a *CallBehavior* action that refers to the behavior described by the "HandlePayment" activity. The translation yields one elementary net for each activity, graphically depicted in figure 6. Names and graphical layout have been added to the figure to help the reader track the transformation.

3.4 Composing Diagrams

The first translation step has allowed to build an IPN type for each diagram of the original specification. Additionally we have built a trace that gives for each UML behavior of the original specification the name of the IPN type produced and the set of links to other behaviors that need to be resolved.

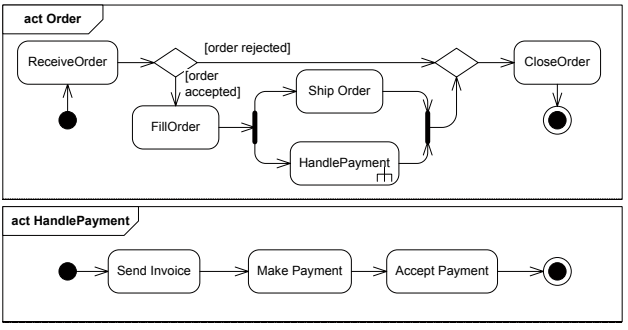


Fig. 5. An example adapted from the UML standard, p.357

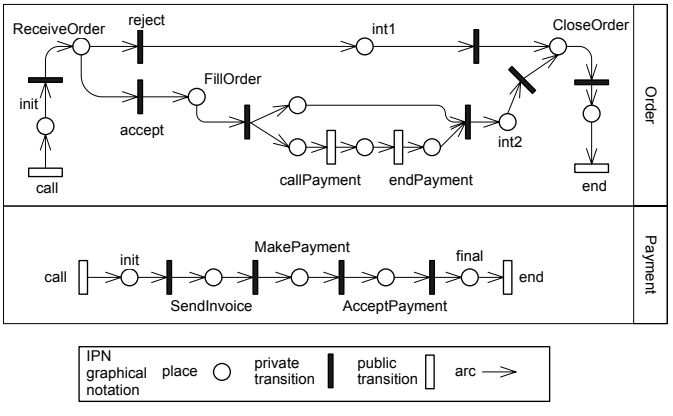


Fig. 6. IPN elementary net types obtained by application of the translation rules

Inter-Diagram Links: The information contained in a link depends on its nature:

- For send event type links, we have the target UML object instance and the name of the transition that sends this event (i.e. noted t in Fig.3).
- For receive event type links, we have the event and the name of the reception transition (i.e. noted t in Fig.3)
- For CallBehavior actions, we have the target UML Behavior and the name of the transitions corresponding to the call and behavior end (i.e. t and t' in Fig.3)
- Finally for CallOperation actions, we have the target operation name and the target UML object instance as well as the names of the call/return transitions of the IPN.

We then incrementally build more complex scenarios by using different linking strategies.

Diagrams in Isolation: The most basic strategy consists in building "isolated" behaviors from each behavior (see Fig.7 top). For a net type n representing a behavior b, we build $isolated(b)$as a composite type containing a single instance i of type n, two public transitions *call* and *end* that synchronize to $i.call$ and $i.end$ respectively, and one private transition synchronizing to each transition of i mentioned in the links. The public link transitions corresponding to interactions with the environment are made private in the isolated type, thus enabled at will provided local conditions allow it.The states active and passive are also defined, as associating the corresponding state to i. The *isolated* construction thus allows to control the consistency of a diagram in an *uncontrolled environment* setting.

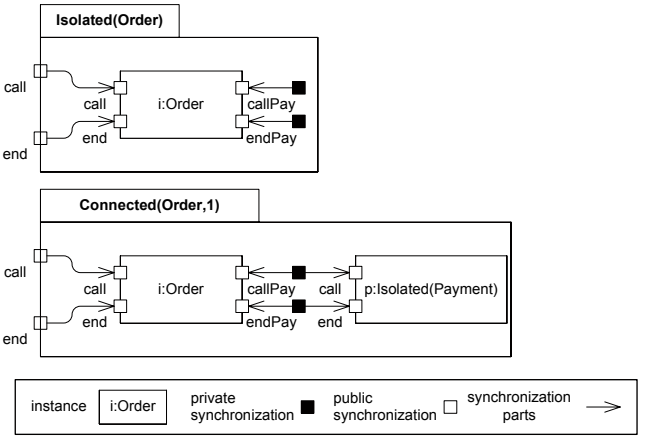


Fig. 7. Composite IPN types produced to analyze composed behaviors

Composing Diagrams: Inductively, we can then build types $connected(b, k)$ (see Fig.7 bottom) corresponding to connections resolved up to depth k, where $isolated(b)$ is equivalent to $connected(b, 0)$. The type $connected(b, k+1)$ contains one instance i of type n (representing b) and for each behavior b' mentioned as a link target, an instance of the $connected(b', k)$ net type. The *call* and *end* transitions are again exported with public visibility, and the links are actually resolved by synchronizing the link transitions of n with the *call* and *receive* of the depth k behavior instances.

UML behaviors can be associated to classifiers or operations. The main use case we have considered is a model where operations may be defined by activity diagrams. Class diagrams are exploited in the following way: when operations of the class are associated with activity diagrams, we construct an IPN type that contains instances of the IPN types corresponding to the activities and "export" the public transitions of the nested activity diagrams. In particular the transitions that allow to start (initial node pattern) and detect the end (final node pattern) of an operation are made visible to connect to CallBehavior of other objects.

When the class contains objects by composition, the corresponding IPN types are instantiated. This gives us a context necessary to determine the target of a CallBehavior or SendEvent action.

3.5 Defining Consistency Checks

One of the main issues when defining properties to verify is that the user (at the UML level) should not explicitly manipulate the formalisms used to specify properties, typically LTL or CTL formulae. This limits the scope of what we can verify, but as we show here, many useful checks can already be defined without user intervention.

Verifying Properties. A model-independent property can be defined regardless of the model instance considered. Typical examples include absence of deadlocks or livelocks, boundedness... These are the easiest to handle as they naturally do not require user input.

We first run a structural bounds computation tool (based on [27]), that produces for each place its structural min/max bounds. This tool scales very well as the check is *structural* so the complexity is linked to the number of places and transitions of the specification, and not to the state space size. This tool allows to detect:

- dead code: [0..0] bounded places indicate unreachable model elements. The interpretation here is that the model element associated to the place is either unreachable (logical design error) or disconnected from the rest of the diagram (modeling bug). In practice, the second interpretation is often correct, as a common error, due to the GUI of UML modeling tools, is to delete elements from a diagram but not from the model;
- unbounded behavior: if a place marking is bounded by $+\infty$ this usually indicates a serious misuse of fork/join constructs. This error was actually raised several times in the models provided by industrial partners of ModelPlex. Neither testing, nor simulation tasks which were working with the same models correctly identified this issue. Simulation actually detected a choke point during performance evaluation and suggested attributing more resources to this faulty activity. Testing would need an infinite test sequence to correctly tag this problem.

When the previous check does not detect unbounded behavior, we can proceed to use model checking based tools for more advanced checks. At this stage we check for absence of deadlocks in the specification. Existence of deadlocks is tagged as an error.

We also check for activity diagrams that their final state is reachable when the initial state is set to "active" (see section 3.3). This is tagged as an error if it is not verified.

The list of verification goals is extensible as we think of new checks interesting for the end-user.

3.6 Conclusions on the Transformation

We thus support the BasicActivities, IntermediateActivities and StructuredActivities packages of the standard, which means our tool handles UML compliance level 2 (see UML2 superstructure [28] for definition of compliance levels).

The weakness of this translation is in the non-determinism of decision nodes, and the absence of any data manipulation. While this may seem a severe restriction, in practice the diagrams we studied (within ModelPlex) were mostly annotated in plain text. They correspond to early phases of design where the logic of the control flow is the focus. Other diagrams we studied were obtained by translation from BPEL, a language for describing the workflow of a business process, where again data manipulation is not the focus. In any case, supporting code level annotations requires extending the standard with a tool specific profile, as the UML standard does not define any data types other than String, Boolean and UnlimitedNatural, or any concrete syntax for actions (e.g. arithmetic).

To overcome this weakness, the transformation could be refined to take into account data, possibly using a CEGAR like approach [9] to limit the induced complexity. This is a perspective at this stage.

The strength of the approach is that it preserves the modularity of the UML specification. Thanks to the concepts of IPN, a UML translation is quite easy to set up, and only one IPN type is produced per UML diagram. When considering several scenarios and a system composed of interacting objects, the fundamental notion of instantiation from object-orientation which is preserved in IPN allows to adequately reuse a model defined in parts. The semantics of IPN are sufficient to capture the concepts of UML behavioral diagrams (sequence, alternative, fork join and parallel behavior, synchronization on events, multiple instantiation...).

Backward correspondence between generated IPN and the original specification is easy to trace since the transformation preserves the structure of UML model. Thanks to this, namespaces are preserved and propagated to IPN objects, thus allowing to trace each transition or place to the corresponding UML model element. THus, when errors are detected in the IPN, it is possible to reconstruct and display a trace in the original UML notation (see figure fig:cs:sap in section 4 where the problems reported refer directly to the UML specification).

4 Behavioral Consistency Checker

We have implemented the approach described in section 3 as a proof of concept in the BCC (Behavioral Consistency Checker) tool.

BCC is designed on top of Eclipse to transparently use formal model checking to provide behavioral consistency checks on UML specifications. Requirements on the tool include that it should need minimal user training, and that the underlying verification technology has to be transparent. To reach these aims, BCC uses UML diagrams as input, and produces easily understandable compiler-style errors and warnings when consistency rules are violated.

This tool was developed as a contribution to the European integrated project ModelPlex[2] (21 industrial and academic partners, 20 M€, 36 months). Within the project, models produced by industrial use case providers are used for several other goals than just verification, such as code generation, requirements traceability and test generation. Thus the investment in the modeling effort is amortized, and any check we can implement that improves the quality of the models is valuable, as the approach is model-centric, and model quality is a goal in itself in this setting. BCC will be integrated in a "model-based simulation, verification and testing (SVT) workbench" that is one of the project deliverables.

The tool is implemented in Java using the EMF framework to parse standard XMI UML models. The transformation is written entirely in Java, rather than using a model transformation engine such as ATL: we had non trivial traceability issues otherwise. Model-checking tools (written in C or C++) are run transparently on the CPN-AMI model-checking platform [20]. Post-interpretation of verification results is again written in Java, using the transformation traces intensively.

A prototype of the tool is available at `http://move.lip6.fr/software/BCC/` and uses services of our CPN-AMI Petri net Framework as a back-end to compute some properties. It currently handles activity diagrams and a subset of state-machine diagrams. It will be completed by the end of ModelPlex (March 2010). We are working on improved class and component diagram handling. Component diagrams allow to have a specific connection topology as the *parts* are class instances. We also work on a sequence diagram translation to IPN. It will allow to control that the sequences describing an expected behavior are indeed executable. Thus it is a means for the user of specifying model-specific properties to check.

Case Study. BCC was used on the case study models from industrial partners in the project. We briefly present here some verification results obtained using BCC.

BCC can be run directly on a XMI UML file, but it is integrated as an Eclipse plugin relying on the EMF/GMF validation service and Eclipse UML2 tools. The user simply sets the scope by selecting one or more UML packages (folders) to be analyzed, and selects the "Validate" action. The model is explored to detect diagrams to be checked, then transformations Preconfigured properties are selected by the designer and then it really works as a push-button tool that can be operated without any knowledge of the underlying techniques.

One study provides a good illustration about the use of such tools. The SAP case study where a small portion is represented in figure 8. This model specifies the activity diagram of a web store system. When payment is to be done, the system must both check credit card and stock. If one condition fails, then it restart the payment procedure (back to *ask client* state).

BCC automatically performs all implemented checks and shows that the specification contained structurally unbounded parts, meaning that, whatever the number of resources provided to execute *check stock* and *check credit*, these transition will still constitute a bottleneck.

[2] `http://www.modelplex.org`

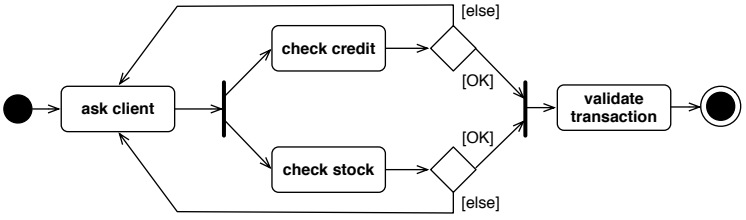


Fig. 8. Portion of the web store system specification

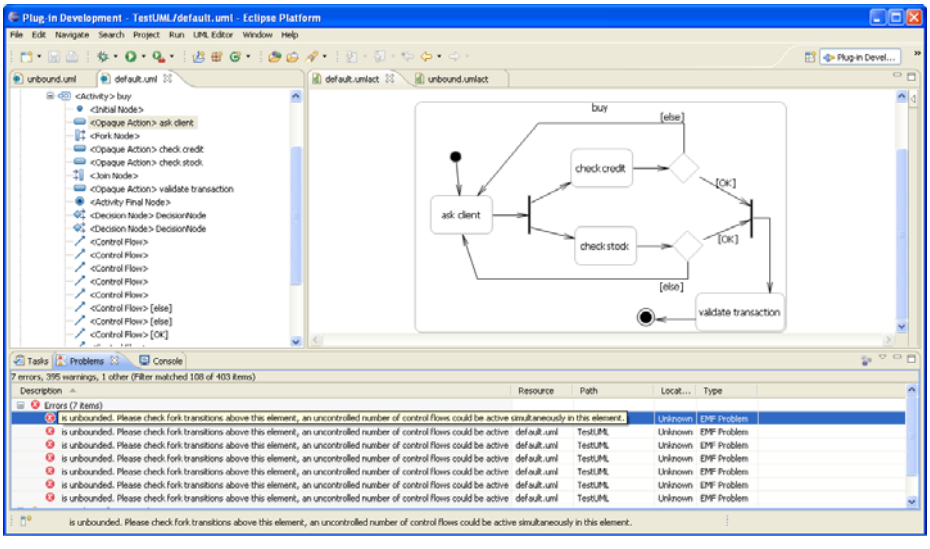


Fig. 9. Window showing bad results for the web store system

Figure 9 show the BCC user interface in Eclipse. The tool is invoked through the contextual menu "validate model" in the model subwindow. Then, the detected problems are reported in the standard "problem view" (below the main window) which usually contains compilation errors. Messages refer to UML objects, double clicking on an error outline the problematic model element in the model browser window.

In parallel with BCC, model-based simulations and test generation for the same system have been performed. Performance oriented simulation detected potential bottlenecks in both *check stock* and *check credit* procedures. However, no explanation of this phenomena was provided. Testing may not be able to capture this bad property of the system, as the bad behavior is exhibited only in infinite traces. Thus model-based verification is a good complement to other model-based validation techniques.

Such model-independent properties (no dead code or boundedness) are of interest to compute on any type of specification. Therefore, a tool like BCC is valuable to help engineers to improve systems early in the software life cycle (in our case, at design level), which is typically a goal of VDE.

5 Conclusion

This paper proposes an approach to extend MDE (Model Driven Engineering) to VDE (Verification Driven Engineering). VDE is MDE together with an intensive use of formal verification techniques to compute properties directly from models, as soon as possible in the software life cycle.

We exemplify this by providing a way to verify automatically some basic properties on UML model without any intervention from an engineer. This example has been implemented in a tool BCC (Behavioral Consistency Checker) ; its experimentation in a project shows that it produces interesting information to engineers to debug their models.

A main point of this paper is the presentation of an original translation mechanism based on Instantiable Petri Nets (ITS), as a basis to express concepts of a higher level modeling language (such as UML). ITS offers a hierarchical way to assemble subsystems or elementary Petri Net components by means of a well-suited set of synchronization mechanisms. ITS also enable efficient verification techniques suitable to tackle large specifications.

So, ITS appear to be an pivot language suitable to:

- Define an operational semantics for a high-level modeling language (here, it was experimented for UML collaboration diagrams and state charts),
- Use this semantics to perform formal verification on the system by means of appropriate techniques such as structural analysis or model checking.

This paper also provides a step-by-step method to elaborate a verification schema based on transformation techniques and patterns associated to the concepts provided in a high-level modeling language. One could use this method, to implement VDE for other languages as soon as its behavioral semantics can be captured by means of appropriate patterns.

References

1. Abrial, J.-R.: The B book - Assigning Programs to meanings. Cambridge Univ. Press, Cambridge (1996)
2. Barkaoui, K., Abdallah, I.: Deadlock avoidance in FMS based on structural theory of Petri nets. In: International Conference on Technologies and Factory Automation (ETFA), pp. 499–510 (1995)
3. Barkaoui, K., Couvreur, J.-M., Dutheillet, C.: On liveness in extended non self-controlling nets. In: DeMichelis, G., Díaz, M. (eds.) ICATPN 1995. LNCS, vol. 935, pp. 25–44. Springer, Heidelberg (1995)

4. Broy, M., Crane, M., Dingel, J., Hartman, A., Rumpe, B., Selic, B.: 2nd UML 2 Semantics Symposium: Formal Semantics for UML. In: Kühne, T. (ed.) MoDELS 2006. LNCS, vol. 4364, pp. 318–323. Springer, Heidelberg (2007)
5. Burch, J., Clarke, E., McMillan, K.: Symbolic model checking: 10^{20} states and beyond. Information and Computation (Special issue from LICS 1990) 98(2), 153–181 (1992)
6. Campos, J., Merseguer, J.: On the integration of uml and petri nets in software development. In: Donatelli, S., Thiagarajan, P.S. (eds.) ICATPN 2006. LNCS, vol. 4024, pp. 19–36. Springer, Heidelberg (2006)
7. Chiola, G., Dutheillet, C., Franceschinis, G., Haddad, S.: On well-formed coloured nets and their symbolic reachability graph. In: Jensen, K., Rozenberg, G. (eds.) Procedings of the 11th International Conference on Application and Theory of Petri Nets (ICATPN 1990); Reprinted in High-Level Petri Nets, Theory and Application. Springer, Heidelberg (1991)
8. Ciardo, G., Lüttgen, G., Miner, A.S.: Exploiting interleaving semantics in symbolic state-space generation. Formal Methods in System Design 31(1), 63–100 (2007)
9. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM 50(5), 752–794 (2003)
10. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (2000)
11. Clarke, E.M., Emerson, E.A., Sifakis, J.: Turing award for their original and continuing research on model checking (2007)
12. Damm, W., Hermanns, H. (eds.): CAV 2007. LNCS, vol. 4590. Springer, Heidelberg (2007)
13. Damm, W., Westphal, B.: Live and let die: Lsc based verification of UML models. Sci. Comput. Program 55(1-3), 117–159 (2005)
14. Dobing, B., Parsons, J.: How UML is used. Communications of the ACM 49 (May 2006)
15. Donatelli, S., Franceschinis, G.: The psr methodology: Integrating hardware and software models. In: Billington, J., Reisig, W. (eds.) ICATPN 1996. LNCS, vol. 1091, pp. 133–152. Springer, Heidelberg (1996)
16. Engels, G., Heckel, R., Küster, J.: Rule-Based Specification of Behavioral Consistency Based on the UML Meta-model. In: Gogolla, M., Kobryn, C. (eds.) UML 2001. LNCS, vol. 2185, pp. 272–286. Springer, Heidelberg (2001)
17. Eshuis, R.: Symbolic model checking of UML activity diagrams. ACM Trans. Softw. Eng. Methodol. 15(1), 1–38 (2006)
18. Gupta, A., McMillan, K., Fu, Z.: Automated Assumption Generation for Compositional Verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 420–432. Springer, Heidelberg (2007)
19. Hailpern, B., Tarr, P.: Model-driven development: The good, the bad and the ugly. IBM Systems Journal 45(3), 451 (2006)
20. Hamez, A., Hillah, L., Kordon, F., Linard, A., Paviot-Adet, E., Renault, X., Thierry-Mieg, Y.: New features in cpn-ami 3: focusing on the analysis of complex distributed systems. In: ACSD, pp. 273–275. IEEE Computer Society, Los Alamitos (2006)
21. Hu, Z., Shatz, S.M.: Explicit modeling of semantics associated with composite states in UML statecharts. Automated Software Engg. 13(4), 423–467 (2006)
22. Huth, M.: Some current topics in model checking. International Journal on Software Tools for Technology Transfer (STTT) 9(1), 25–36 (2007)
23. ISO/IEC 13568. Z formal specification notation — syntax, type system and semantics (2002)

24. Kordon, F., Hugues, J., Renault, X.: From Model Driven Engineering to Verification Driven Engineering. In: Brinkschulte, U., Givargis, T., Russo, S. (eds.) SEUS 2008. LNCS, vol. 5287, pp. 381–393. Springer, Heidelberg (2008)
25. Madhusudan, P. (ed.): Proceedings of the 9th International Workshop on Verification of Infinite-State Systems (INFINITY 2007), Lisboa, Portugal, September 2007. Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers, Amsterdam (2007)
26. Milner, R.: Communicating and Mobile Systems: the Pi-Calculus. Cambridge University Press, Cambridge (1999)
27. Murata, T.: Petri nets: Properties, analysis and applications. In: Proceedings of the IEEE, NewsletterInfo: 33Published as Proceedings of the IEEE, April 1989, vol.77(4), pp. 541–580 (1989)
28. OMG. Unified Modeling Language: Superstructure - Version 2.1.2 formal/07-11-02. OMG (November 2007)
29. SAE. Architecture Analysis & Design Language V2 (AS5506A) (January 2009), http://www.sae.org
30. Thierry-Mieg, Y., Poitrenaud, D., Hamez, A., Kordon, F.: Hierarchical set decision diagrams and regular models. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 1–15. Springer, Heidelberg (2009)
31. Yao, S., Shatz, S.M.: Consistency Checking of UML Dynamic Models Based on Petri Net Techniques. In: CIC 2006: Proceedings of the 15th International Conference on Computing, Washington, DC, USA, pp. 289–297. IEEE Computer Society, Los Alamitos (2006)
32. Zhao, Q., Krogh, B.: Formal verification of statecharts using finite-state model checkers. IEEE Transactions on Control Systems Technology 14(5), 943–950 (2006)

Cross-Document Dependency Analysis for System-of-System Integration

Syed Asad Naqvi[1,2], Ruzanna Chitchyan[1], Steffen Zschaler[1], Awais Rashid[1], and Mario Südholt[2]

[1] Lancaster University, Lancaster, UK
naqvis@comp.lancs.ac.uk, r.chitchyan@lancaster.ac.uk, szschaler@acm.org, awais@comp.lancs.ac.uk
[2] DCS, École des Mines de Nantes, Nantes, France
Mario.Sudholt@emn.fr

Abstract. Systems-of-systems are formed through integration of individual complex systems, often not designed to work together. A number of factors can make this integration very challenging which often leads to catastrophic failures. In this paper, we focus on three major classes of system-of-system integration problems: managerial independence, interface incompatibility, and component-system complexity. We then present an aspect-oriented requirements description language (RDL) which uses natural language analysis capabilities to reason about dependencies across the documentation of the constituent systems of a system-of-systems. The aspect-oriented compositions in the RDL also facilitate specification of cross-document constraints and inconsistency resolution strategies, which can be used for deriving proof obligations and test cases for verification and validation of the emergent behaviour of a system-of-systems. We showcase the capabilities of our RDL through a case study of a real-world emergency response system. Our analysis shows that the querying and composition capabilities of the RDL provide valuable support for reasoning across documentation of multiple systems and specifying suitable integration constraints.

1 Introduction

As software systems become increasingly pervasive in our daily lives, we are seeing the emergence of a new class of systems, that of, systems-of-systems (SoS) [1]. These SoS are at least an order of magnitude greater in complexity than their conventional counterparts. Examples of such systems are airport management systems, airline alliances, healthcare systems, disaster response and recovery systems, etc. However, for a SoS to function effectively, all the constituent systems need to work towards a common overall goal. This is not always the case given that a SoS comes into being as a consequence of emergent rather than pre-planned requirements. Even in case of pre-planned integration, unforeseen problems can arise owing to the dichotomy between individual system goals and those of the SoS. A recent example of such dichotomy can be observed in the

C. Choppy and O. Sokolsky (Eds.): Monterey Workshop 2008, LNCS 6028, pp. 201–226, 2010.

Heathrow Airport Terminal 5 problems, due to incompatibilities between the baggage handling system and the check-in system. The problem was further compounded by the fact that, as is usual in SoS, both constituent systems were under the control of different organisations—the goals of a SoS have to be understood and maintained across such organisational and system administration boundaries.

Reconciling the goals of a SoS with those of its constituent systems is non-trivial—a SoS is an ultra large-scale system [2] that comes into being by the collaboration of a number of systems that may belong to different domains. This multi-domain characteristic of SoS is often referred to as heterogeneity (or diversity) [3,4,5,6,7] and has been identified as one of the basic properties of SoS. Another common characteristic in SoS is distribution [1,4,5,8,9,10] which entails geographical distribution of the collaborating systems forming part of a SoS. Furthermore, because of its scale, a SoS affects and is also directly affected by external forces such as political, economic, social, and legal factors.

The need for taking a holistic view of the SoS means that the stakeholders from the constituent systems need to communicate with each other and understand each other's perspectives. The heterogeneity and distribution inherent to a SoS make such communication extremely challenging. Furthermore, the influence of external factors needs to be understood when reasoning about the overall behaviour of a SoS.

The key challenge for stakeholders in a SoS is, therefore, to understand or learn about other systems (besides their own) and external influences [4,11]. A large portion of this learning takes place by studying the various description and specification documents of other systems as well as legal guidelines, operational procedures, etc. Especially during the process of integrating different systems to form a SoS the system specifications, operational procedures, user manuals, business case documents, test reports, etc. of the different systems need to be analysed, perhaps cross-examined, to find conflicts and also opportunities for optimisation. The scale of this task, similar to the extreme complexity of a SoS, is beyond what can reasonably be achieved by a team of engineers without scalable automation support.

Substantial advances have been made in the field of natural language analysis in the past two decades that can be exploited to study large document sets. Similarly, with the recent emergence of aspect-oriented requirements engineering (AORE) techniques [12,13], there are new reasoning mechanisms available to both study and specify constraints that crosscut multiple system specifications. In this paper, we present an aspect-oriented requirements description language (RDL) [14] and show how we utilise the semantics of the natural language itself to both explore and capture cross-document dependencies, as well as conflict resolution strategies, in RDL-based aspect-oriented composition specifications. Since the composition specifications are based on natural language semantics, they facilitate intentional reasoning about cross-document dependencies; that is, reasoning about the stakeholders' intentions as expressed in the document text. This allows us to better understand stakeholders' requirements for SoS

and identify and resolve any discrepancies between the overall goals of the SoS and those of its constituent systems.

The remainder of the paper is structured as follows. In Sect. 2 we discuss three main classes of SoS integration problems. Section 3 presents the natural language analysis concepts pertinent to our RDL, its semantic queries and semantics-based composition specifications which are, respectively, used to identify cross-document dependencies of interest and specify resolution policies. Section 4 details a case study from the emergency-related communications domain to show how the RDL can be used to tackle the three classes of integration problems highlighted in Section 2. We discuss some of the open research problems in Sect. 5 and how the RDL's capabilities may be extended to tackle these. Section 6 reviews related work and Sect. 7 concludes the paper.

2 Integration Challenges in Systems-of-Systems

The key factors underpinning the integration problems in SoS fall intro three main classes:

1. The collaborating systems in a SoS are managerially independent;
2. The evolution trajectory of the processes and interfaces of individual systems does not account for potential future collaboration or communication with other systems in a SoS context;
3. The individual systems are often, themselves, extremely complex and therefore difficult to understand and integrate with other systems.

We next discuss each of these classes in detail, with the help of some well-known examples of SoS integration failures.

2.1 Managerial Independence

Our notion of Managerial Independence is that different systems in a SoS are being managed as well as operated by different groups of people. Though managerial independence is a key characteristic of SoS [15], it also poses two major integration challenges:

– *Decision-making and maintenance of an overall vision at the SoS level:* The managers of individual systems are often unaware of the protocols and procedures of other systems so there is a lack of overall vision and control. Furthermore, the decisions required for integration and co-working have to be made by the cooperation and agreement of all the concerned parties. Obtaining this agreement can often be non-trivial even impossible. An example of this is the European Union. In June 2008 the people of the Republic of Ireland (which represent only one percent of the EU population) voted to reject the Treaty of Lisbon. For the treaty to come in to force, it requires ratification from all 27 countries in the EU. Even though 25 countries had ratified the treaty, rejection from just one country meant that the treaty could not come into effect. Such problems faced by decision-makers in the SoS context have also been highlighted in [4].

– *Managing a large-scale socio-technical system:* A SoS is inherently a socio-technical system that encompasses the activities of the humans as well as the hardware and software systems in its boundary. The individual systems that collaborate in a SoS will often continue to operate independently in order to provide some functionality independent of the SoS. Participating in a SoS puts additional stress on the human part of the individual systems. People may have to concentrate on two different lines of work. Concerns about errors arising from excessive workload and the interactions of various human roles in a SoS have also been put forward as a major issue [16].

An example of SoS integration failure due to Managerial Independence is the failure of Heathrow Terminal 5 (T5) on its opening day. T5 was built at a cost of 4.3 billion pounds. The period from project commencement to terminal opening day was approximately five and a half years. The last six months were an operations readiness period in which the final preparations before opening—staff training and terminal systems and process testing—were to take place [17].

The terminal fell seriously short of expectations on its opening day. Thousands of people faced a chaotic situation when the terminal's baggage handling system stopped working. Sixty eight flights were cancelled [17] and thousands of people were deprived of their luggage.

The examination of what went wrong on the opening day reveals that the cause of the problems was a systems integration and coordination problem between the two main agencies responsible for operating T5: the British Airport Authority (BAA), which is the agency that built the terminal, and British Airways (BA), the (only) airline operating from the terminal. Both of these agencies were working together and were responsible for the operations of the terminal. The lack of communication, coordination, and integration between BAA and BA prior to and on the opening day was the root cause of the following problems:

– Many amongst the BA staff were not familiar with the equipment that they were supposed to operate. This equipment had been provided by BAA [17,18].
– BA claimed that they were unable to completely test the software systems under their control because BAA did not finish construction of the building in time [19].
– The baggage system failed because BAA had responsibility for the system at baggage check-in while BA had responsibility for the baggage loading part of the system [17]. The system finally shutdown because the rate of baggage check-in was higher than the rate of baggage loading [17].
– There was no crisis management system setup between BA and BAA at the terminal level [17].

Colin Matthews (chief executive of BAA) noted the lack of cooperation between the management of BA and BAA as the main cause of the problems at Terminal 5. In a statement [17] he said:

> "However well the airport operator and the airline operator, BA, are working it is also vital that the two are absolutely integrated and together. I think that during the construction of Terminal 5 that appeared

to be the case. Around about or just prior to the opening of T5 it seems that that togetherness deteriorated. It is that togetherness that allows you to cope with the issues that arise on the day."

2.2 Incompatible Interfaces

Systems evolve over the course of their lifetime to meet new demands. Often this evolution takes place in a stove-pipe-like narrow domain. The protocols and interfaces that the system defines are often not meant for communication, collaboration, or evolution beyond this specific domain. This lack of foresight in the systems' architecture places extra strain [20] when these systems are required to collaborate with other systems. There is also the principle of encapsulation [21] in systems engineering to consider. This principle calls for systems to be closed off from the outside and to hide their implementation and inner working. In order for individual systems in a SoS to collaborate, they may need to break the principle of encapsulation and allow invasive access [22] to their inner working from other systems [3].

The interoperability challenge [23] facing the various emergency response agencies in the US is an example of this class of problems. The emergency-response communication systems of various agencies and jurisdictions often evolve without taking into consideration the need to communicate with other agencies and jurisdictions.

An example of SoS integration failure, due to Incompatible Interfaces, is the chaotic rescue operation that took place after the Air Florida Flight 90 disaster in Washington DC [24]. On 13 January 1982 the Air Florida Flight 90 crashed into the Potomac River shortly after takeoff. A number of federal, state, and local emergency response agencies took part in the rescue effort. The rescue effort was greatly hampered due to the lack of compatibility between the communication systems of the different agencies. The different emergency-response agencies had evolved their communication systems independently without taking into consideration the requirements of integrated rescue efforts with other agencies. Consequently, the rescue personnel from different agencies could not communicate with each other [24].

2.3 Complexity of the Collaborating Systems

The complexity of a system has a direct impact on its understandability. Given that the constituent systems in a SoS are often highly complex themselves, integration problems arise because of one's inability to fully comprehend the workings of the individual systems. As a result, it is often not possible to predict the behaviour of the participating systems within the context of the SoS. As the participating systems in a SoS are often interdependent on each other, it may take only one system to malfunction for the entire SoS to break down. Such reliability problems in highly complex systems have been discussed in [4].

An example of SoS integration failure due to the complexity of collaborating systems is the loss of the Mars Climate Orbiter. The Mars Climate Orbiter

(MCO) started its journey to Mars on December 11, 1998 from Cape Canaveral, Florida. Its mission was to gather information about the Martian atmosphere and to act as a relay station for the Mars Polar Lander Mission. On September 23, 1999 it was lost while trying to setup an orbit around Mars. The MCO was designed, developed, and operated by the collaboration of two agencies—NASA and Lockheed Martin Astronautics (LMA) [25]. The cause of the crash was in a piece of software that was producing its results in English units instead of metric units. But according to Dr. Edward Weiler, NASA's Associate Administrator for Space Science:

> "The problem here was not the error, it was the failure of NASA's systems engineering, and the checks and balances in our processes to detect the error. That's why we lost the spacecraft." [26]

The review panel that conducted the analysis of the failure found that the best processes and standards for software development had been followed. As noted in [27], the problem was not that of methodology. The failure instead arose from the "sheer complexity" of the system . The different consituent systems of the space craft were so complex that errors within them could not be detected despite the use of rigorous means and best practices.

Of course, the above three classes of problem are not orthogonal. As shown in Table 1, all three SoS integration failure cases: T5, Air Florida Flight 90 and MCO, exhibit multiple classes of failures.

Table 1. Multiple causes of SoS integration failures

Class of Failure / *SoS*	Managerial Independence	Incompatible Interfaces	Complexity of Collaborating Systems
Heathrow T5	√		√
Air Florida Flight 90	√	√	
Mars Climate Orbiter	√	√	√

In this paper, we propose natural-language document-processing techniques to help uncover the above classes of problems. Any such cross-document analysis must provide three capabilities:

- Querying over multiple documents in a fashion that accounts for diverse writing styles and usage of different terms to refer to the same concepts;
- Specification of constraints across multiple documents in order to explicitly capture conflict resolution strategies and, subsequently impose them, through derivation of proof obligations or suitable test cases for the SoS;
- Automation support to aid the engineers and stakeholders in such querying and constraint specification during SoS integration.

In the next section, we discuss RDL—our natural-language based requirements description language—and its support for these capabilities.

3 Requirements Description Language (RDL)

Our RDL, detailed in [14], is based on the observation that the natural language used in systems' documentation already reveals semantics that can be used as a basis for both analysis of dependencies and specification of compositions that relate process specifications that span multiple documents or specify constraints that resolve inconsistencies. For this, we utilise the vast body of work on natural language grammar, semantics, and natural language processing (NLP) [28,29,30]. The RDL is based on scalable tool support from the WMatrix NLP engine [31], which has been shown to provide high accuracy: up to 97% for part-of-speech analysis and 93% for semantic analysis of English language texts [32].

Any document to be analysed using the RDL capabilities needs to be annotated with suitable grammatical and semantic information. This annotation is fully automated via WMatrix as the relevant information can be extracted directly from the document text. The composition specifications use these annotations as a basis of *semantic queries* which can be used both for uncovering dependencies across documents as well as specifying points of interest to specify crosscutting constraints and resolution strategies across documents. Naturally, this requires human input to encode relevant domain knowledge. The RDL is, therefore, not a substitute for a human analyst but instead a tool that can aid the complex task of studying reams of documentation and specifying constraints, resolution strategies or operational procedures that crosscut multiple documents.

The RDL was conceived as an aspect-oriented mechanism [33] to provide support for more modular representation and analysis of natural language-based requirements texts. It lends itself to analysis of SoS integration problems (and subsequent encoding of resolution strategies) as it uses:

- natural-language grammar to identify the grammatical elements that are prominent in conveying the semantics of the natural-language sentences, and, thus, are relevant for studying cross-document dependencies;
- natural-language semantics for expressing the meaning of the identified grammatical elements, and various ways of referencing them when querying the various documents under consideration;
- aspect-oriented composition mechanisms to specify crosscutting constraints and conflict resolution strategies. These constraints and strategies can be subsequently used as a basis for proof obligations or test cases for verification and validation of overall SoS behaviour.

In the following sub-sections, we first present the NLP fundamentals underpinning the RDL. We then present how this semantic and grammatical information is used as a basis for the semantic queries. Finally, we discuss how aspect-oriented composition mechanisms are utilised to specify cross-document constraints and conflict resolution strategies.

3.1 Fundamentals of the RDL

The RDL is build upon two main pillars: semantic and grammatical fundamentals. Each of these is further discussed below.

Semantics Foundations. In this work, semantics refers to the meaning expressed in text. In particular, we draw on the:

- principles of similarity of meaning (i.e., synonymy) for main parts of speech groups (i.e., nouns, verbs, adjectives, and adverbs) since mostly these are the groups undertaking the main grammatical functions in a clause.
- We also use some properties of word formation (i.e., word morphology) that allow reduction of word forms.
- We propose to utilise the domain specific knowledge of entities and their dependencies captured in ontology building.
- Finally, we propose to utilise a number of semantic categories, i.e., groupings of words in accordance with their relevance to a particular classification scheme, e.g., per domain, such as words describing human activities, or Animal-related words, etc.

Each of these concepts is briefly presented below.

Synonyms are different morphological forms (i.e., words) with same sense (i.e., used with a similar or same meaning) [34, pp 70–71]. Synonymous words are generally interchangeable. For instance, in an online auction system, "to place a bid" means the same as "to make a bid", or "to bid"; or "concurrently" is the same as "in parallel" and "goods" is the same as "products". Synonymy is widely used in natural language, thus synonyms must be recognised and regarded as the carriers of the same semantics for successful natural language text analysis and understanding. For the RDL this implies that a reference to an element via its synonyms is supported.

Word Form Reduction (Lemmatisation): A given word normally has a number of possible forms, for instance "to bid", "bidding", "bids" are all about bidding and are all formed by modifications of the basic word form "bid". This most basic form of a word is called lemma. Lemmatisation (or reduction of the word to its most basic form) is widely used in natural language processing in order to simplify natural language text analysis. For the RDL, we take the view that a lemma is representative of a single part of speech only. For instance, if the text contains "bidder" as well as "to bid" we will have two lemmas, one for "to bid" as a verb and another for "bid" as a noun. Consequently, in the RDL compositions a reference to the verb's lemma will not be confused with that of the noun's [35].

From the perspective of RDL design, using lemma-based referencing allows a narrower scoping of the reference (i.e., only to all forms of the specific word), while synonym-based referencing allows for a wide scoping—to all words with a given meaning.

Ontology: an ontology is a schematisation of knowledge of a domain, representing the concepts of the domain, properties of each concept, and the relationships between the concepts. Ontologies have a number of uses, including building common understanding of information, representation, analysis, and reuse of domain knowledge, etc. However, each ontology is built to answer a specific set of questions, and, for this reason, the same domain can be represented via a number of differing ontologies [36].

For the RDL, the ontologies can be used for retrieving ontology-supported information from the requirements documents (as is done in such disciplines as information management, semantic web [37]). For instance, if the "is-a" hierarchy is represented, it could be used to identify all classes and subclasses of a given concept relevant to a given composition specification; if "part-of" relationship is provided, the ontology can help in identifying all constituents of a given stakeholder concern, etc.

Semantic Categories for Nouns and Verbs: a number of categorisations for major grammatical categories have been developed. For instance, in accordance with Quirk [34] nouns can be grouped into 5 main categories for *concrete nouns* denoting physical world entities (e.g., grass, hill, etc.), *abstract* which refer to abstract notions (e.g., happiness, friendship, etc.); *states* and *properties* reflecting mental and corporeal states and properties (e.g., hunger, pleasure, etc.); *activities* which are nouns describing activities (e.g., sale, decision, etc.). Each of these can be sub-categorised into smaller, more specialised groups.

Similarly, verbs can be categorised in accordance with their specific properties. We use one of such verb groupings for RDL, as discussed below.

Verb classifications and Role-based Interaction Patterns. Several prominent results in linguistics [29,30,38] have shown that there is a clear link between the meaning of the words and their grammatical behaviour. Such a link can be illustrated via a simple experiment presented in [38]: two English speakers were asked about the correct use of an archaic English verb *gally* which was used in whaling. They were presented with a sentence "Sailors gallied the whales." Then they were asked if use of *gally* in the sentence "Whales gally easily." is appropriate. The speaker who thought that gally meant "see", believed that it was incorrect, as we don't say "Sailors saw the whales. Whales see easily." On the other hand, the speaker who thought gally to mean "frighten", believed that it was correct to say "Whales gally easily", as it is correct to say "Whales frighten easily".

In line with the above experiment, Dixon [29] suggests that the varying grammatical behaviour of verbs is the result of the differences in their meaning. Thus, using this principle, he groups verbs into several semantic categories. The verbs in each semantic category require the same type of participating roles. For instance, all Giving type verbs require a Donor, Gift, and Recipient roles, as in "Allan (Donor) gave the keys (Gift) to Peter (Recipient)"; all Attention verbs require a Perceiver and an Impression role, as in "The instructor (Perceiver) witnessed the accident (Impression)", etc. In some cases certain roles can be omitted, or understood from the context or from the most common use of the verb.

In our work we use the semantic categories of [29] as basis for identifying the types of relationships between concerns and, deriving composition operators.

We observe, that generally in natural language, the semantics of action-type dependencies (denoted by action-operators, or actions as per [13,39]) are expressed by verbs or verb phrases. But, in accordance with Dixon's verb classification [29] there is only a limited set of broad meanings of verbs, thus, there

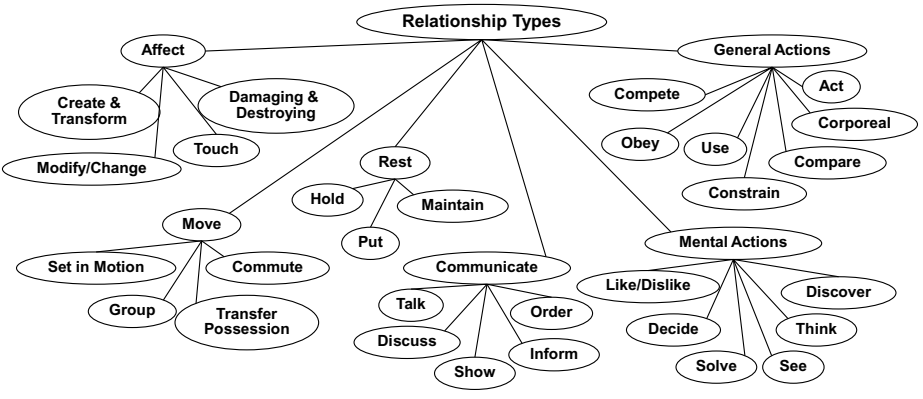


Fig. 1. Requirements analysis specific rearrangement of Dixon's verb classes

must only be a limited set of broad dependency types (and, correspondingly, operators of action type) that correspond to the verb categories. There are 63 verb classes suggested by Dixon [29]. Having reviewed these categories for suitability from the perspective of composition semantics for requirements [40], we have identified the set of verb categories presented in Figure 1 for use in RDL.

Grammar Foundations. In natural language, a sentence is considered the highest-ranking unit of grammar [34]. Thus, we utilise a sentence and its main constituents—subject, verb, and object—to form RDL elements. One or more sentences make up a Requirement.

A *Requirement* is a description of a service the stakeholders expect of the system, the system behaviour, and constraints and standards that it should meet [41]. The requirements specified using the RDL are annotated natural language sentences. Each requirement may contain one or several clauses [34]. Each clause contains sub-elements for subject, relationship and optionally for object(s).

One or more requirement elements are encapsulated within a *concern* which is a module for encapsulating requirements pertaining to a specific matter of interest (e.g., selling and account management). A concern can be simple (containing only requirements), or composite (containing other concerns as well as requirements). Each concern is identified by its name.

A *subject* is the entity that undertakes actions described within the requirement clause. Subject in our RDL corresponds to the grammatical subject in the clause. In order to support interaction (i.e., composition) specifications involving a subject denoted with different words representing the same semantics, a set of synonymous definitions must be provided. These synonyms could be provided either through a standard synonym dictionary or per project through project specific dictionaries.

An *object* is the entity which is being affected by the actions undertaken by the subject of the requirement sentence, or in respect with which the actions

are undertaken. Object in our RDL corresponds to the grammatical object in the clause. Usage and properties of an Object are very similar to those of the Subject. However, in a requirement there could be several objects associated with (affected by) a single subject.

Relationship depicts the action performed (state expressed) by the subject on or with regards to its object(s). Relationships can be expressed by any of the verbs or verb phrases in natural language. Using Dixon's verb categories [29], we classify relationships into a set of types (the second level nodes in Fig. 1, such as Move, etc.) and their more specific sub-types (the 3rd level nodes in Fig. 1, Set in Motion, etc.). The various relationship categories derived from Dixon's verb classification are detailed in [14,40].

It must be noted, that we do not suggest that ALL semantics of a requirement are reduced to Subject-Relationship-Object constructs (SRO). Indeed, elsewhere we are looking at such element of requirements as degree of importance (i.e., which requirements are more urgent compared to others) or quality satisfaction, among others. However, we suggest that SROs are the main elements with respect to which the rest of the requirement semantics are formulated. Thus, SROs are the elements which participate in relations with other requirements, and are qualified, constrained or otherwise defined by both single requirement semantics, and the inter-requirements dependencies. Such semantics and dependencies can be both queried and captured in the RDL compositions, as discussed below and detailed in [14].

Tool Support for RDL. The annotation of the subjects, relationships, and objects is supported via a set of links on top of the tags assigned by the Wmatrix part-of-speech (POS) tagger. The links are inserted by matching flexible patterns of POS tags. These patterns have been identified by linguists using a combination of linguistic knowledge and corpus evidence. For example, a simple rule to link a verb to its object is as follows:

```
N*o[.] (RR*/RG*/XXn3) VVN*v[.]
```

This matches the sequence 'Noun' (`N*`), followed by between 0 and 3 possibly negated 'adverbs' (`RR*/RG*/XX`), followed by a past participle 'verb' (`VVN`). In the case of a match, the noun is marked as the object of the verb. The subjects, verbs and objects are marked explicitly by Wmatrix along with the result of lemmatisation.

The annotation of verb types is realised via a mapping from the Wmatrix verb categories onto an RDL-specific tagset. In some cases the large classes of Wmatrix words were directly compatible with the RDL verb classification, for instance, the verbs of domain for Movement, Location, Travel and Transport in Wmatrix largely correspond to the RDL verbs of Motion type. On the other hand, there are semantic classes in the Wmatrix (e.g., Education, Time, etc.) tagset which have no correspondence to that of the RDL verbs tagset and their contents have been mapped to the RDL verb classes on an individual verb by verb basis.

This annotation has been automated and is quite fast. In a recent test with a file of 56,031 words (237 pages), it took about 20 minutes to complete the initial annotation [42]. However, the quality of the annotation is significantly influenced by the quality of the input document, since every sentence of the input document is annotated with the RDL. Our POS-based patterns work rather well for well structured sentences with clearly defined subjects, verb, and objects, achieving roughly 80% of accuracy[1]. These patterns perform poorer when very long, multi-level nested sentences or grammatically incorrect sentences are used. Similarly, the quality of verb class annotation suffers if the verbs used in the text have not been covered in the RDL tagset.

3.2 Semantic Queries

The query expressions in the RDL are called semantic queries, since they select concerns/requirements on the basis of the semantics of (parts of) these concerns or requirements. The queries can use all kinds of annotations provided by the RDL, including the SRO, verb types and semantics (e.g., relationship.type= "Modify"), concern names, etc.[2] It should be noted that a requirement may have several sentences, but if one clause of one of its sentences matches the specified semantics, the requirement will be relevant for this query. Benefits of the semantic queries are twofold: firstly, we avoid syntactic matching (e.g., based on specific labels) in the queries and associated composition specifications, thus avoiding unintended element matching. Instead, queries and associated compositions are specified based on the semantics of the requirements. Secondly, it ensures that any compositions specified are semantically justified, rather than arbitrarily provided by an analyst. We provide examples of the semantic queries when discussing the composition specifications next.

3.3 Semantics-Based Composition of Concerns

A composition rule in the RDL comprises three elements: *Constraint, Base* and *Outcome.* Each of these elements has an operator attribute and a semantic query expression. The query expression can select whole concerns or individual requirements from within concerns. A concern is selected if the *concern* keyword is used in the query, otherwise requirements are selected.

The *Constraint* element specifies what constraint/restriction will be placed on some requirements and what action must be taken in imposing these constraints (e.g., a conflict resolution strategy to address mismatch across constituent system specifications). The restriction that this element imposes is defined in its query expression. The action that needs to be taken is defined by the constraint operator.

[1] This is an estimate based on our experiments, but is not formally validated yet. Currently we cannot provide exact time measurement for automated and manually corrected specification generations.

[2] The RDL has an XML-based syntax, which is used to automatically annotate the natural language text. For simplicity, in this paper, we omit the XML-based syntax.

An example of two Constraint elements are shown in Fig. 2 (note we omit the XML-based RDL syntax for simplification):

- Here the first Constraint element has a query *(relationship= "assign" and object = "liaison station")* stating that all requirements where a something is *assigned* to be a *liaison station* should be selected.
- The second Constraint element has a query *(subject= "liaison station" and relationship = "contact" and object= ("Assistant Section Manager" OR "Section Manager"))* stating that all requirements where liaison stations contact the Assistant Section Manager or Section Manager should be selected.

  Note, these queries do not specify where physically such requirements should be located and do not refer to any additional characteristics of these requirements. They directly point to the relevant meaning of requirements: assigning as a liaison station and the liaison station contacting the (Assistant) Section Manager.
- The "create" operator in the first Constraint specification implies that the roles of Agent, Manipulation Entity, and Target are relevant for this composition. From the Constraint query we can identify the "liaison station" as the Target; that is, something will be made into such a station.
- The "correspond" operator in the second Constraint specification implies that the roles of Speaker, Addressee, Medium and Message are relevant. From this query we also know that the Speaker is the liaison station who contacts the Addressee—here the (Assistant) Section Manager.

The *Base* element reveals the set of requirements that are affected by the elements selected in the Constraint element's query. The operator in the Base element depicts the temporal or conditional dependency between the requirements selected by the Base element query and those of the Constraint query.

An example of a Base element is shown in Fig. 2:

- The base query *(relationship="activate" and object= "RACES net")* notes that all requirements where *activation* of *RACES net* is mentioned are to be selected.
- The base operator *(meets)* denotes that immediately upon realisation of Base query, the relevant Constraint query requirement(s) must be applied. In this case, as soon as RACES net is activated, a *"liaison station"* must be assigned and the *liaison station* must *contact* the *(Assistant) Section Manger.* In addition, we can see that the RACES net will act in the Manipulation role, whereby some Actor will create a liaison station from a RACES net station as per the operator of the first Constraint.

It is worth noting that since the RDL is based on a symmetric model, it is possible to choose any set of concerns (using semantic queries) as Constraint and any other set as Base. The same requirement may be selected by a Constraint query in one composition, and by a Base query in another. We do not discuss the automation of the actual composition process and its subsequent analysis in this paper (details are available in [14,43] using ideas from [39]).

Finally, the *Outcome* element defines how imposition of constraint requirements upon the base set of requirements should be treated. For instance, the outcome element may specify a set of requirements that must be satisfied as post-conditions upon application of the Constraint; in this case the respective operator, such as *satisfy* will be used with that query. Unlike for Base and Constraint elements, the semantic query of the Outcome element can be empty, if no additional requirements/concerns are affected due to the Base and Constraint element interactions. In this case the *ensure* operator can be used to indicate that though there is no additional Output query, the relationships between Constraint and Base must be ensured.

4 Case Study: Reasoning about System-of-System Integration

Having discussed the three classes of SoS integration problems in Sect. 2 and presented the main elements of the RDL in Sect. 3, we now illustrate how the RDL can facilitate handling of the integration problems.

RDL works on requirements documents. Therefore, to show how RDL can be used for SoS, we would ideally possess some requirements documents where real issues of SoS integration have arisen. Unfortunately, such requirements documents are typically not available in the public domain. In the absence of requirements documents from known cases of SoS integration problems, we sampled and analysed arbitrary documents from a domain in which cases of such integration problems exist. For this paper, we selected the domain of emergency-related communications-based on the Air Florida disaster. This domain is attractive since a large number of related documents on this subject are freely available at a dedicated portal[3]. From this domain we randomly selected two: one detailing the radio communication procedures for the Virginia Emergency Net (VEN) that supports emergency communications for the state of Virginia, USA[4], and the other detailing the same for the Radio Amateur Civil Emergency Service (RACES) for the counties of Carroll, Grayson and the City of Galax in Virginia and California[5]. We treated these documents as prototypes of requirements documents as they may be used for SoS in the domain of emergency communication. We then applied the RDL to these documents analysing them for examples of managerial independence, interface incompatibility, and complexity issues. In the following, we discuss one such example for each of these integration problem classes. In the following examples, while discussing the RDL, we leave out the XML annotations for better readability.

It is interesting to note that for each case, our use of RDL essentially follows the same three steps:

1. Using natural-language processing, we produce a formal encoding of the two documents in RDL.

[3] http://www.safecomprogram.gov/SAFECOM/
[4] http://www.w4ghs.org/vensop2.pdf
[5] http://www.w4ghs.org/Twin_County_SOP.pdf

2. Using RDLs query mechanism, we search both documents for statements that may indicate potential problems arising when the two systems need to collaborate. We then check with experts and stakeholders to understand whether these statements indeed represent problems and, if so, what should be done about them.
3. We use the RDL's composition mechanism to encode any resolutions to the problems we found or to explicitly encode any conflict-resolution schemes that are already implicitly present in the documents.

4.1 Resolving Managerial Issues for SoS Integration

Since the VEN and RACES will need to cooperate to handle emergency-related communications, we need to understand who will activate the emergency procedures and how these two organisations will interact. In order to identify information related to activating these organisations, we can query the RDL-annotated document texts for the relevant information. Thus, we:

1. Find where the activation topics are treated in the documents by finding the activation related verbs, such as *activate* and its synonyms *make active, set in motion, set off, turn on, trigger, get going, trigger, prompt, initiate. Activate* is a verb of Set in Motion type which has defined roles for the *Causer* (normally taking the subject function) who sets into motion a *Moving Object* (normally taking the object function) with the optional noun phrase to specify the *Locus* (i.e., where?) role.
2. Identify what actors are filling in the appropriate roles with these action verbs:
   (a) Roles in RACES:
      i. Upon notification from the OEM or E-911 Director (Causer, though not directly defined) the *plan* (Moving Object) will be **set in motion.**
      ii. *EOC locations* (Moving Object) may be **activated** and covered with Amateur Radio (Causer) but net control should be posted outside this busy area.
      iii. An Amateur Radio Hospital Volunteer (Causer) will **activate** this *station* (Moving Object).
      iv. The Twin County RACES Emergency net (Causer) **operates** as the *Virginia/Carolina Training and Information Net* and meets every Sunday at 3:00 pm (1500 hrs) on the Fishers Peak repeater (145.130 - 600 with a tone of 103.5).
   (b) Roles in VEN:
      i. Either the SM or SEC (Causer) shall **activate** *The Virginia Emergency Nets* (Moving Object).
3. Check if there are any parallels in the activation procedure and the involved actors. In the above example:
   – The Causers are: OEM, E-911, Amateur Radio, Amateur Radio Hospital Volunteer, Twin County RACES Emergency Net, SM, SEC.

- The Moving Objects are: plan, EOC locations, station, Virginia/Carolina Training and Information Net, Virginia Emergency Nets.
- In points (2a: i and iv) we find that OEM, E-911 and Twin County RACES Emergency net may have some common responsibilities in activating the RACES net in an emergency. From 2b: i, we can see that the SM (Section Manager) or SEC (Section Emergency Coordinator) will activate the Virginia Emergency Net. Thus, there needs to be a protocol of interaction between these bodies to coordinate emergency handling in Virginia.

4. Since there is a need to unify the activation of these emergency bodies, there needs to be a managerial procedure to coordinate these bodies. We check if such a procedure already exists by looking at cross-references and/or collocations of the above-identified bodies in our two different documents.

   In Virginia Emergency Net document we find:

   - "VEN/D 3620 kHz (7105 & 14103.3 kHz alt) Packet, Pactor; Digital Operations - NON-RACES OPS ASM/ASEC/D"
   - "VEN/D RACES 3543 kHz (7105 & 14103.3 kHz alt) Packet, Pactor, Digital Operations - ONLY REAL RACES OPS ASM/ASEC/D"

   Here we also find that ASM/ASEC/D corresponds to Assistant Section Manager/Assistant Section Emergency Coordinator for Digital Operations. Thus, from this document we have identified that the Assistant Section Manager is assigned to the communications involving RACES operations.

   In RACES document we find:

   - "Liaison stations to the following National Traffic System nets will be assigned (Old Dominion Emergency Net-3947) (Old Dominion Emergency Net 7240) (***Virginia Emergency Net*** 3910)."

   Here we have identified that a liaison station will be assigned to communicate with the Virginia Emergency Net.

   Thus a procedure of communication between these two systems has become clearer: it transpires that the Assistant Section Manager or Section Manager will be responsible for managing the communication between VEN and RACES via a RACES liaison station.
5. Finally, we assert this communication procedure by defining a specific composition (Fig. 2).

Thus, by using the synonym-based querying of the RDL we were able to identify the areas in the input documents where issues related to activation were discussed. We then identified entities that carry out same roles for the activation process in different documents and were able to consider their relations to each other in the management of the two systems and their co-working. Based on these considerations, we then defined a composition to assert a managerial process for the interaction of the two systems.

4.2 Addressing Incompatible Interfaces in SoS Integration

The second problem in SoS integration, discussed in Sect. 2.2 of this paper, relates to the incompatibility of interfaces between systems. We will now consider

```
Composition: VEN_RACES_Communication
Constraint: create
    operator: enable
    query: (relationship= "assign" and object = "liaison station")
Constraint:
    operator: correspond
    query: (subject= "liaison station" and relationship = "contact" and
           object= (Assistant Section Manager" OR "Section Manager"))
Base:
    operator: meets
    query: (relationship="activate" and object= "RACES net")
Outcome:
    operator: ensure
```

Fig. 2. Composition for Communication Procedure between VEN and RACES

how such incompatibility requirements can be identified and resolved with our RDL-based approach.

The previously discussed integration failure in the case of Air Florida Flight 90 was caused due to the use of different communication frequencies by different emergency teams. Let us now check if such a problem may arise in integration of the VEN and RACES systems. To do this, we need to:

1. Verify that the frequencies listed against each of the nets are consistent across documents. Here we use domain knowledge about the format of describing nets and their frequencies by listing the frequency immediately before/after the name of the net—for example, "Old Dominion Emergency Net 7240"—without explicitly using the term "frequency" or its synonyms in the description.
2. Look up where use of frequencies is explicitly mentioned by looking up the term *frequency* and its synonyms, such as *Hz* and checking with stakeholders/managers that these are correct for each net.

In these two example documents we have a number of references to nets and tier respective frequencies, such as:

- In the RACES document there are references to:
  - Old Dominion Emergency Net 7240
  - Virginia Emergency Net 3910
- In Virginia Emergency Net document there are references to:
  - 3543 kHz (7105 & 14103.3 kHz alt) RACES
  - 7240 kHz Alt ... ODEN (i.e., Old Dominion Emergency Net)

Thus the reference from the RACES document to ODEN frequencies is consistent with that from the Virginia Emergency Net.

The next reference from RACES document is to Virginia Emergency Net ***3910.*** However, in the document on Virginia Emergency Net there is no mention of a frequency of 3910. Instead, it refers to ***3543*** kHz (***7105*** & ***14103.3*** kHz alt)

frequencies for RACES. This obviously is an area that needs clarification with the stakeholders:

- do the documents have the correct *different* frequencies listed, or
- is there an incompatibility in the specification and these documents must use the *same* frequency? If there is an incompatibility here, what is the correct frequency to be used?

For this example we assume that the communication protocols used require that the stations use the same frequency to communicate. In which case, we have identified an integration problem for which a resolution decision must be made. Let us assume that the specification in the document for the Virginia Emergency Net is chosen as the correct one. We can now define a composition specification, as shown in Fig. 3, asserting that the values of frequencies for VEN listed in the VEN document (frequency.value=doc.VEN and object="VEN") will dominate over those listed in the RACES document (frequency.value=(doc.RACES and object= "VEN")). This composition will ensure harmonised resolution of this issue in future requirements.

```
Composition: VEN_RACES_Frequency_Harmonisation
Constraint:
    operator: modify
    query: frequency.value=doc.VEN and object="VEN"
Base:
    Operator: concurrent
    Query: frequency.value=(doc.RACES and object= "VEN")
Outcome: ensure
```

Fig. 3. Composition for Interface Harmonisation between VEN and RACES

4.3 Support for Handling Complexity in Systems' Documentation

We have previously discussed the issue of complexity in understanding behaviours of the systems to be integrated. From the perspective of requirements analysis, this complexity manifests in the need to identify and understand the behaviours of interest from a large volume of written documentation (since our discussion relates to the documents written in natural language). Thus, here the complexity mainly manifests itself in the need to treat large volumes of information.

The utility of our approach lies in the ability to:

1. *identify and separate concerns of interest* from the rest of the documentation and
2. *define localised compositions* to assert a particular set of rules/interaction resolutions that crosscut multiple documents and/or concerns within these documents.

This aspect-oriented basis of the RDL provides inherent support for modular (studying a particular concern of interest in isolation from other concerns in a system, in this case a SoS) and compositional reasoning (reasoning about the interdependencies and relationships amongst concerns to understand the emergent behaviour, in this case the behaviour of the SoS) [44].

An example of such modular and compositional reasoning was illustrated in Sect. 4.1, whereby a particular behaviour related to activation was studied independently from the rest of the documents' contents. This behaviour was considered in tandem with the set of entities required to carry out the given behaviour in different documents and the relationships of these entities were also considered. In Sect. 4.2, we also presented an example composition definition which was used to harmonise differences between the two input documents.

The example documents in this case study are largely unstructured natural language texts. The RDL compositions can also be used to define crosscutting behaviour across sets of more structured requirements artifacts, such as use case specifications. For instance, a behaviour related to sending message will be discussed in a number of use cases detailing the RACES and VEN systems. A single localised composition may be used to specify that an encryption protocol should be employed at any point when a message is sent. Such a composition specification will not require any change/rewriting of the existing use cases. Yet, when the relevant use cases are viewed/analysed in an appropriately tooled environment, the additional detail on encryption use will be incorporated across the relevant use case steps. Further details on the crosscutting nature of the RDL compositions is provided in [14,45].

5 Discussion

In Sect. 4, we have discussed how our RDL can be used to study the three major classes of integration failures in SoS: managerial independence, incompatible interfaces and complexity of constituent systems. Significant benefit can also be derived from our RDL-based approach when we consider a set of documents containing natural language requirements for which a large number of economic, social, political factors of heterogeneity must be observed, constraints enforced and priorities maintained. In these cases the semantics-based queries of the RDL can assist in direct identification of relevant points in the documents where, for instance, a particular policy is discussed, etc. Moreover, when needed, a locally defined composition specification can be used, for instance, to enforce a new policy, or introduce the behaviour of new system into a broad set of existing operations specification documents.

However, there are a number of other SoS-characteristic problems that can occur during the analysis and cross-examination of specification documents. Below we discuss some of them as well as some potential extensions of our RDL approach to address them:

1. *Different languages:* Because of the geographic distribution of collaborating systems in a SoS, their specification documents may be written in different

languages e.g. one set of specification documents may be in English and the other in German.

To handle such differences the RDL approach can be furnished with a cross-language analysis support. Such a cross-language support is indeed feasible, since all the grammatical and semantic features of the RDL are manifest in the vast majority (if not all) human languages whereby an entity (subject) carries out some activity (verb) on or in respect with other entities (objects). Moreover, the verb classes identified by Dixon, and used in the RDL, are also largely language-independent, as discussed in [29].

2. *Regional syntax:* It is a well known fact that the same language may have regional dialects. These dialects can become so well rooted as to make their way in to the written word and thus establish themselves as different versions of the same language. A well-known example is the difference between the US and UK versions of the English language. For example two different words—soccer and football—refer to the same sport in the US and UK respectively. However, the same word football refers to two different sports in the US and UK.

   The RDL approach can be furnished with support to identify the dialects and automatically establish correspondences between relevant entities. For this we will need to build a language corpus for each relevant dialect and provide a set of algorithms which would identify the dialect used in each document from the natural language clues. Clues such as use of words/expressions unique to a given dialect (e.g., *"fall semester"* in US English); spelling and/or grammar peculiarities (e.g., "behavior vs. behaviour"), frequencies of particular word use, etc. can be utilised for the purpose.
3. *Domain specific syntax:* This problem can be further divided into two subcategories.
   (a) Domain specific jargon in specifications of the collaborating systems: certain words like "Tympanum" from the "Anatomy" domain may not be understandable to non-experts.
   (b) Different domain specific meaning of the same word: for example, the word "Delta" means different things in Geology and Mathematics.

   The first of these problems can be resolved by providing domain specific lexicons and/or ontologies, where required. However, building these can be a substantial effort in itself. This can be facilitated (or substituted) with a set of algorithms which could use the previously discussed techniques (i.e., most frequently used concepts, unique words, and combinations etc.) to identify the general domain of the document and to provide a summation of relevant term occurrences and/or definition from a set of sample documents of that domain (e.g., by obtaining these from the Web).

   The resolution of the second of these problem can already be supported via word sense disambiguation techniques used in NLP, such as realised in the Wmatrix [31] tool.

A number of other problems, such as *varying formats of specification* documents (e.g., some written using use case specifications, some using viewpoint-based

templates); *inconsistent detail* (e.g., some systems may be specified in great detail while others may have almost no documentation); *misinterpretation of data* (e.g., in one specification a mean value is used as representative, while in the other system a mode), etc. will also arise in distributed heterogeneous SoS. While these require further research and development effort, it could be relevant to note that the RDL would be well positioned in supporting such problem resolutions since:

- The RDL is based on natural language characteristics and does not require any other specific format or restrictions;
- It has been shown that the RDL approach is amenable to automation [14], and already has automated support for a number of processing activities [14,43];
- The RDL uses the semantics of natural language to identify relevant portions in the different system specification documents, allowing domain experts to focus only on details relevant to their work;
- The RDL compositions are able to support localised specifications of cross-document dependencies/constraints.

6 Related Work

Ultra large scale (ULS) systems have been defined in [2] as "A system at least one of whose dimensions is of such a large scale that constructing the system using development processes and techniques prevailing at the start of the 21st century is problematic." These futuristic systems will exhibit characteristics not unlike SoS today. These characteristics include decentralized control, conflicting requirements, heterogeneity, evolution, failures of parts of the system, and erosion of the people/system boundary etc. [2] Problems similar to the ones facing the requirements analysis of SoS have also been raised for ULS systems. These include finding compatibility, redundancy, inconsistency, emergent properties, in requirements and reasoning about requirements in the presence of uncertainty and ambiguity [2]. Therefore, the approach presented in this paper, may also be relevant for ULS systems.

A number of approaches have been proposed in recent years focusing on the use natural language processing and information retrieval techniques for analysis of crosscutting concerns. The EA-Miner tool [45,46] uses the WMatrix toolset to mine for crosscutting concerns in natural language requirements specifications. EA-Miner is a tool that provides integrated support for the RDL, generating RDL specifications from the mined (and, subsequently edited) requirements model.

Other relevant works include Theme/Doc LSA [47] and Repertory Grid [48]. Theme/Doc LSA uses the Latent Semantic Analysis technique [49] to build a concern-requirement matrix. In the matrix concerns correspond to terms while requirements correspond to documents used in the analysis. The LSA algorithm is then used to identify relevant concern-requirements associations. Threshold values can be set to filter out associations that are less pertinent or not of

interest. However, too low a threshold value can lead to cluttered concern-requirement association graphs while too high a value can lead to more sparsely populated graphs. We have found that a hybrid approach, combining LSA with NLP, yields better results as the NLP-based analysis can be used to identify elements of interest which can then be subject to an LSA analysis for identifying relationships [50].

The work of Niu and Easterbrook [48] is based on the Repertory Grid technique from Psychology which aims at capturing how people construct mental models of objects in their environment. Using this technique, one can study how two constructs may be similar or different in a particular document set, hence identifying the contributions of specific tasks to high-level system goals. However, in contrast to our approach, this technique requires structured requirements as input and the construction of the grid is not automated. However, its integration with the RDL can yield fruitful results by allowing one to study the mental model of a SoS (for instance, based on observations from ethnographic studies) from the perspective of the various stakeholders of the constituent systems. This can facilitate a more top-down analysis of SoS integration challenges compared to the bottom-up documentation-based analysis supported by our approach.

7 Conclusion and Future Work

The conception and development of a SoS poses a number of challenges, not least due to the fact that SoS are created opportunistically owing to some social or business need. Since the constituent systems are often not designed to work together in the first place or, when they are they remain under the control of different autonomous organisations, incompatibilities are almost inevitable owing to the inherent complexity of the individual systems, their managerial independence, and past (often divergent) evolution trajectories. Reasoning about the overall behaviour of a SoS is, therefore, non-trivial in the presence of such diversity and heterogeneity. We cannot escape the fact that mostly such reasoning is based on reading reams of documentation about the individual systems-almost 80% of system specifications remain in natural language. The simple search facilities in word processing or file rendering systems are not able to relate concepts that may have been referred to using different terms. Furthermore, it is not possible, without specialist tools that create vendor lock-ins and require use of specific specification notations, to specify constraints across the documentation.

Our RDL presents a solution to these challenges. It is based on well-established natural language processing concepts and can be deployed across domains. The semantic queries in the RDL work on rich information clues already inherent in natural language specifications hence making it possible to relate concepts as well as specify composition rules that work on this natural language semantic basis. This, in turn, means that the composition rules are resilient to changes in documentation structure. Most significantly, the RDL approach can be utilised with any documentation written in natural language. Our case study has shown that the approach can uncover integration issues in real-world SoS and can be used to specify suitable resolution strategies.

We do not consider the RDL to be the final solution to these challenges. Instead we see it as a stepping-stone towards scalable mechanisms for reasoning about SoS integration issues. In Sect. 5 we have identified a number of future work paths for this work. They will be our focus in the long-term. In the short-term, we aim to apply the RDL to further case studies of real-world SoS to gather more empirical data about its efficacy based on a larger corpus of SoS examples.

We are also currently working to develop a more detailed, hierarchical taxonomy of expected SoS integration problems and faults. Such a taxonomy may help to focus the attention of the SoS engineer towards finer grained problems. These problems include, but are not limited to, technical incompatibilities, quality of service issues, and regulatory compliance problems. Once developed, this taxonomy can be used as the basis of a library of RDL query templates. These templates can assist the SoS engineer by lessening the work of writing queries for expected SoS problems. The SoS engineer will of course have to customise or instantiate the templates for the particular problem at hand. For example, a query template can be written to help the SoS engineer query the system's documentation to find out what kind of communication protocols are used or supported by the system. The template method is expected to be useful in performing quick, cursory comparisons of different requirement and specification documents. Detailed analysis of requirement documents is still expected to require writing unique, standalone queries in RDL.

Acknowledgements

This work is supported by EC FP6 project, AMPLE: Aspect-Oriented Model-Driven Product Line Engineering and the EC FP7 project DiVA: Dynamic Variability in Adaptive Systems. Awais Rashid is also supported by a Chair Regionale by the Pays de la Loire Regional Government in France.

References

1. Sage, A.P., Cuppan, C.D.: On the systems engineering and management of systems of systems and federations of systems. Information, Knowledge, Systems Management 2(1), 325–345 (2001)
2. Northrop, L., Feiler, P., Gabriel, R.P., Goodenough, J., Linger, R., Longstaff, T., Kazman, R., Klein, M., Schmidt, D., Sullivan, K., Wallnau, K.: Ultra-Large-Scale Systems: The Software Challenge of the Future. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (July 2006)
3. Boardman, J., Sauser, B.: System of systems: The meaning of of. In: IEEE Int'l. System of Systems Conf., April 2006, pp. 118–123 (2006)
4. DeLaurentis, D., Callaway, R.: A system-of-systems perspective for public policy decisions. Review of Policy Research 21(6), 829–837 (2004)
5. DeLaurentis, D.: Role of humans in complexity of a system-of-systems. In: Duffy, V.G. (ed.) HCII 2007 and DHM 2007. LNCS, vol. 4561, pp. 363–371. Springer, Heidelberg (2007)

6. Jamshidi, M.: System of Systems Engineering: Innovations for the 21st Century. John Wiley & Sons, Inc., Chichester (2008)
7. Keating, C., Rogers, R., Unal, R., Dryer, D., Sousa-Poza, A., Safford, R., Peterson, W., Rabadi, G.: System of systems engineering. EMJ – Engineering Management Journal 15, 36 (2003)
8. Sage, A.P.: Conflict and risk management in complex system of systems issues. In: IEEE Int'l. Conf. on Systems, Man and Cybernetics (2003)
9. Eisner, H.: RCASSE: rapid computer-aided systems of systems engineering. In: 3rd Int'l. Symposium of the National Council of System Engineering (NCOSE), pp. 267–273 (1993)
10. Kotov, V.: Systems of systems as communicating structures. Technical report, Hewlett Packard Computer Systems Laboratory Paper HPL-97-124 (1997)
11. Popper, S.W., Bankes, S.C., Callaway, R., De-Laurentis, D.: System of systems symposium: Report on a summer conversation. In: 1st System of Systems Symposium (2004), http://www.potomacinstitute.org/academiccen/SoSSummerConversationreport.pdf
12. Baniassad, E.L.A., Clements, P., Araujo, J., Moreira, A., Rashid, A., Tekinerdogan, B.: Discovering early aspects. IEEE Software 23(1), 61–69 (2006)
13. Rashid, A., Moreira, A., Araujo, J.: Modularisation and composition of aspectual requirements. In: International Conference on Aspect-Oriented Software Development (AOSD), pp. 11–20. ACM, New York (2003)
14. Chitchyan, R., Rashid, A., Rayson, P., Waters, R.W.: Semantics-based composition for aspect-oriented requirements engineering. In: International Conference on Aspect-Oriented Software Development (AOSD), pp. 36–48. ACM, New York (2007)
15. Maier, M.: Architecting principles for systems of systems. Systems Engineering 1(4), 267–284 (1998)
16. Madni, A.M., Sage, A.P., Madni, C.: Infusion of cognitive engineering into systems engineering processes and practices. In: IEEE Int'l. Conf. on Systems, Man and Cybernetics, October 2005, pp. 960–965 (2005)
17. House of Commons Transport Committee: The opening of Heathrow Terminal 5. Twelfth Report of Session 2007-08. HC 543, Published on 3 November 2008 by authority of the House of Commons London (2008), http://www.publications.parliament.uk/pa/cm200708/cmselect/cmtran/543/543.pdf (December 16, 2008)
18. BBC News: What went wrong at heathrow's T5? (March 2008), http://news.bbc.co.uk/1/hi/uk/7322453.stm (December 16, 2008)
19. Thomson, R.: British airways reveals what went wrong with Terminal 5 (May 2008), http://www.computerweekly.com/Articles/2008/05/14/230680/british+airways+reveals+what+went+wrong+with+terminal.htm (December 16, 2008)
20. Ellison, R.J., Goodenough, J., Weinstock, C., Woody, C.: Survivability assurance for system of systems. Technical report, CMU Software Engineering Institute, CMU/SEI-2008-TR-008, ESC-TR-2008-008 (May 2008)
21. Rechtin, E.: Systems Architecting. Prentice-Hall, Upper Saddle River (1990)
22. Benavides Navarro, L.D., Südholt, M., Douence, R., Menaud, J.-M.: Invasive patterns for distributed programs. In: Meersman, R., Tari, Z. (eds.) OTM 2007, Part I. LNCS, vol. 4803, pp. 772–789. Springer, Heidelberg (2007)
23. Smith, B., Tolman, T.: Can we talk? Public safety and the interoperability challenge. National Institute of Justice Journal, 17–21 (April 2000)

24. The Office for Interoperability and Compatibility (OIC), The Department of Homeland Security: The system of systems approach for interoperable communications (2008), http://www.safecomprogram.gov/NR/rdonlyres/FD22B528-18B7-4CB1-AF49-F9626C608290/0/SOSApproachforInteroperableCommunications_02.pdf (last visited, January 2008)
25. Mars Climate Orbiter Mishap Investigation Board: Phase I report (November 1999), ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf
26. NASA: Mars climate orbiter official website (1998), http://mars.jpl.nasa.gov/msp98/news/mco990930.html
27. Marshall, S.: Software engineering: Mars climate orbiter, http://www.vuw.ac.nz/staff/stephen_marshall/SE/Failures/SE_MCO.html
28. UCREL: UCREL semantic analysis system (USAS). Lancaster University, UK (2006), http://www.comp.lancs.ac.uk/ucrel/usas/
29. Dixon, R.M.W.: A Semantic Approach to English Grammar, 2nd edn. Oxford University Press, Oxford (2005)
30. Levin, B.: English verb classes and alternations: a preliminary investigation. The University of Chicago Press, Chicago (1993)
31. Rayson, P.: WMATRIX. Lancaster University (2007), http://www.comp.lancs.ac.uk/ucrel/wmatrix/
32. Sawyer, P., Rayson, P., Cosh, K.: Shallow knowledge as an aid to deep understanding in early phase requirements engineering. IEEE Trans. Software Eng. 31(11), 969–981 (2005)
33. Filman, R.E., Elrad, T., Clarke, S., Akşit, M.: Aspect-Oriented Software Development. Addison-Wesley Professional, Reading (2004)
34. Quirk, R., et al.: A Comprehensive Grammar of the English Language. Longman, London (1985)
35. Francis, W.N., Kučera, H.: Frequency Analysis of English Usage: Lexicon and Grammar. Houghton Mifflin, Boston (1982)
36. Noy, N.F., McGuinness, D.L.: Ontology development 101: A guide to creating your first ontology. Technical report, Stanford Knowledge Systems Laboratory and Stanford Medical Informatics, Technical Report KSL-01-05 and SMI-2001-0880 (2001)
37. Fensel, D., Hendler, J.A., Lieberman, H., Wahlster, W.: Spinning the Semantic Web. The MIT Press, Cambridge (2002)
38. Hale, K.L., Keyser, S.J.: A View from the Middle. MIT, Center for Cognitive Science, Cambridge (1987)
39. Moreira, A., Araujo, J., Rashid, A.: Multi-dimensional separation of concerns in requirements engineering. In: 13th IEEE Int'l. Conf. on Requirements Engineering (RE 2005), pp. 285–296 (2005)
40. Chitchyan, R., Rashid, A.: Tracing requirements interdependency semantics. In: Workshop on Early Aspects (held with ASOD 2006), Bonn, Germany (2006)
41. Sommerville, I.: Software Engineering, 2nd edn. Addison-Wesley, Reading (2004)
42. Sampaio, A., Rashid, A., Chitchyan, R., Rayson, P.: EA-Miner: Towards automation in aspect-oriented requirements engineering. Transactions on Aspect-Oriented Software Development 3, 4–39 (2007)
43. Waters, R.W.: MRAT – the multidimensional requirements analysis tool. Master's thesis, Computing Department, Lancaster University (October 2006)
44. Rashid, A., Moreira, A.: Domain models are NOT aspect free. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 155–169. Springer, Heidelberg (2006)

45. Chitchyan, R., Pinto, M., Rashid, A., Fuentes, L.: COMPASS: composition-centric mapping of aspectual requirements to architecture. Transactions on Aspect-Oriented Software Development 4, 3–53 (2007)
46. Sampaio, A., Chitchyan, R., Rashid, A., Rayson, P.: EA-Miner: a tool for automating aspect-oriented requirements identification. In: 20th IEEE/ACM Int'l. Conf. on Automated Software Engineering (ASE 2005), pp. 352–355. ACM, New York (2005)
47. Kit, L.K., Man, C.K., Baniassad, E.: Isolating and relating concerns in requirements using latent semantic analysis. SIGPLAN Not. 41(10), 383–396 (2006)
48. Niu, N., Easterbrook, S.M.: Analysis of early aspects in requirements goal models: A concept-driven approach. Transactions on Aspect-Oriented Software Development 3, 40–72 (2007)
49. Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval. Addison-Wesley, Reading (1999)
50. Alves, V., Schwanninger, C., Barbosa, L., Rashid, A., Sawyer, P., Rayson, P., Pohl, C., Rummler, A.: An exploratory study of information retrieval techniques in domain analysis. In: 12th Int'l. Software Product Line Conf. (SPLC 2008), Washington, DC, USA, pp. 67–76. IEEE Computer Society, Los Alamitos (2008)

Performance Analysis of AADL Models Using Real-Time Calculus*

Oleg Sokolsky[1] and Alexander Chernoguzov[2]

[1] Department of Computer and Info. Science, University of Pennsylvania, Philadelphia, PA, U.S.A.
[2] Honeywell, Fort Washington, PA, U.S.A.

Abstract. Architecture Analysis and Design Language (AADL) captures both platform and software architectures of embedded systems in a component-oriented fashion. Properties embedded in an AADL model enable several high-level analysis techniques. In this work, we explore how to perform analysis of end-to-end timing characteristics of an AADL model using Real-Time Calculus (RTC). We identify properties of AADL models that are necessary to enable such analysis and develop an algorithm to transform an AADL model into an RTC model. We use the proposed technique to identify analyze the infrastructure for sensor network architecture.

1 Introduction

Architecture Analysis and Design Language (AADL) [5,13] is a modeling framework for embedded systems. It captures both platform and software architectures of an embedded system in a component-oriented fashion. A typical AADL model captures system threads, their mapping to processors, and connections between threads that represent flows of control and data through the system.

Because systems are specified at a high level, without much behavioral detail, AADL models can be developed relatively early in the development cycle and used for the evaluation of design alternatives. Therefore, there is a great need for analysis techniques that can be applied to AADL models in order to establish global properties of the models, such as schedulability, reliability, latency of data flows through the system, etc.

Many such analysis techniques are available, and analysis models can be extracted from AADL models. For example, fault tree models [7] can be extracted from AADL models equipped with error modeling information. Resource allocation decisions are analyzed in [3]. In [9], the authors consider simulation of AADL fragments. Rate monotonic schedulability analysis can be applied to AADL models with periodic tasks in a straightforward ways, and in [14] we presented an approach for the schedulability analysis of more complicated AADL models.

* Research is supported in part by AFOSR FA9550-07-1-0216 and NSF CNS-0720703 grants and a grant from Honeywell.

C. Choppy and O. Sokolsky (Eds.): Monterey Workshop 2008, LNCS 6028, pp. 227–249, 2010.

Real-Time Calculus (RTC) is a high-level analysis technique that allows to compute quantitative estimates of end-to-end timing of stream-processing hard real-time systems. In this work, we show that it is possible to extract an RTC model from an AADL model and perform the analysis of relatively large models. The structure of the RTC expression directly reflects dependencies between AADL components, making the translation easy to understand and implement in a tool.

As an illustration of the proposed analysis technique, we consider an architecture for a wireless sensor network from the industrial automation domain. The architecture includes sensor nodes and gateways that connect the wireless network to the wired network that connects it to various data consumers. We show how the RTC-based analysis helps us understand performance of the architecture as more sensor nodes and gateways are added. The case study also allows us to assess the scalability of the proposed analysis method.

The paper is organized as follows. Section 2 offers brief overviews of AADL and RTC. Section 3 describes the translation of AADL into RTC. Section 4 introduces the case study and shows analysis results. Finally, we conclude by offering a discussion of future work in Section 5.

2 Background

2.1 AADL Overview

AADL is an architecture description language for distributed embedded systems [5]. In addition to graphical and textual syntax and high-level operational semantics defined in [13], AADL incorporates a modeling methodology, formulated in [4]. This work incorporates property sets defined in version 2 of the AADL standard.

AADL modeling and analysis is supported by an open-source extensible development environment OSATE [6] based on Eclipse. The open nature of OSATE facilitates the development of analysis tools by providing an API to explore and navigate the model and present analysis results in a uniform way. A number of analysis tools are available for AADL models, offering simulation, schedulability analysis, resource budgeting, etc. To the best of our knowledge, there are no available performance analysis tools for AADL.

Components. The main modeling notion of AADL is a *component*. Components can represent a software application or an execution platform. A component can have a set of externally accessible *features* and an internal implementation that can be changed transparently to the rest of the model as long as the features of the component do not change. Implementation of a component can include interconnected subcomponents. The features of a component include data and event ports and port groups, subroutine call entries, required and provided resources. Data ports represent sampled communication and are unbuffered. Event and event data ports represent message passing. Each input event or event data port has a FIFO buffer associated with it. Interacting components can have their

features linked by event, data, and access connections. In addition, application components can be bound to execution platform components to yield a complete system model. Properties, specific to a component type, can be assigned values that describe the system design and can be used to analyze the model. We will discuss properties relevant to the RTC transformation in Section 3. Component types are illustrated in Figure 1. Different component types are shown as different shapes, according to the standard. Solid lines represent connections, while double lines represent bindings.

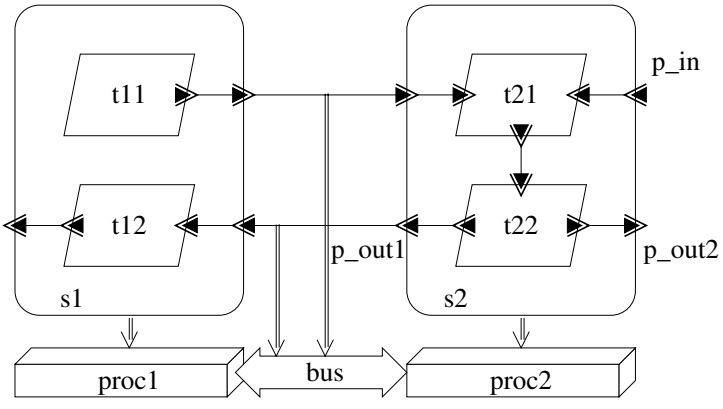


Fig. 1. Simple AADL model

Execution platform components include *processors*, *buses*, *memory blocks*, and *devices*. Properties of these components describe the execution platform. Processors are abstractions of hardware and the operating system. Properties of processors specify, for example, processing speed and the scheduling policy. Buses can represent physical interconnections or protocol layers. Their properties identify throughput and latency of data transfers, data formats, etc.

Application components include threads, processes, and systems. *Threads* are units of execution. Each thread has an associated *semantic automaton* that describes thread states and conditions on transitions between thread states. A thread can be halted, inactive, or active. An active thread can be waiting for a dispatch, computing, blocked on resource access, or preempted. A thread can also be recovering from a fault or in the state of non-recoverable error. Properties of the thread specify computation requirements (execution time ranges) and deadlines in active states of the thread, dispatch policy, etc. Threads are classified into periodic, aperiodic, and sporadic threads. They differ in their dispatch policies and their response to external events. A periodic thread is dispatched by a timer, while aperiodic and sporadic threads are dispatched by the arrival of events, as discussed below. A *process* component describes the scope of memory protection: all components in a process share the same address space. As processes are transparent to our analysis technique, we do not discuss them any

further. A *system* component is a unit of composition. It can contain application components along with platform components, and specifies bindings between them. Systems can be hierarchically organized.

In an AADL model, threads can be bound to processors and connections can be bound to buses. Threads bound to the same processor are scheduled according to the scheduling policy of the processor and collectively utilize the resource offered by the processor components. Similarly, data transmitted along the connections bound to the bus component impose a load on the respective communication resource.

Figure 1 shows a simple AADL model that we will use throughout the paper to illustrate features of the language, the transformation into RTC, and RTC analysis. The system component contains two processors connected by a bus, and two software subsystems. Each of the subsystems is bound to a separate processor. Subsystem *s1* contains one periodic and one aperiodic thread and subsystem *s2* contains two aperiodic threads. The system has one input event data port and one output event data port. Threads communicate via event data ports. Note how features of a component – in this case, input and output event data ports – are mapped by connections to features of its subcomponents.

Connections. AADL connections can connect ports of two components within the same system, or of a system and one of its subcomponents. A sequence of connections, connected by ports at their end points, forms a *semantic connection*. Each semantic connection has an *ultimate source* and *ultimate destination*. Ultimate sources and destinations can be thread or device components. Starting from an ultimate source, a semantic connection follows connections up the component containment hierarchy via the outgoing ports of enclosing components, includes one "sibling" connection between two components, and then follows connection down the component hierarchy until it reaches the ultimate destination. One of the semantic connections in Figure 1 is between threads *t11* in the system *s1* and *t21* in the system `s2`. This connection contains three syntactic connections and is bound to the bus component. When a sporadic or aperiodic thread is the ultimate destination of an event connection, it is dispatched by the arrival of an event via that connection. If another event arrives while the thread is executing, it is queued at the port. The next dispatch occurs as soon as the thread becomes idle. By contrast, periodic threads, which are dispatched by a timer, treat events in the queues of their event ports as data.

Similarly, semantic access connections describe resources required by a thread that is the ultimate source of an access connection. A resource that serves as the ultimate destination of an access connection is typically a data component. Properties of access connections specify concurrency control protocols for shared resources.

Modes. AADL can represent multi-modal systems, in which active components and connections between them can change during an execution. Mode changes occur in response to events, which can be raised by the environment of the system or internally by one of the system components. For example, a failure in

one of the components can cause a switch to a recovery mode, in which the failed component is inactive and its connections are re-routed to other components. The AADL standard prescribes the rules for activation and deactivation of components during a mode switch.

In this work, we do not consider multimodal systems. Analysis described here can be applied to each global system mode separately. The OSATE toolset supports this way of analysis by offering a separate single-mode AADL model for each global mode of a multimodal AADL model. Direct support of multiple modes may be achieved by extending RTC model with event sequence automata considered in [17]. Event sequence automata will represent changes to the AADL model during a mode switch. This approach is left for future work, however.

2.2 Real-Time Calculus

RTC [15,1] is a formalism that is based on the network calculus [10]. Our presentation of RTC closely follows that of [17], which give a much more detailed exposition of the approach. RTC is used for system-level performance analysis of stream-processing systems with hard real-time constraints. Modular performance analysis based on RTC [19] represents the embedded system as a collection of abstract processing components, which process incoming events and require a certain amount of resource in order to perform this processing. Such a component can represent either computation or communication in the system. When representing computation, an abstract component can represent, for example, a real-time task. The task is dispatched for execution when an event arrives and requires some amount of time - typically represented as best-case and worst-case execution times - in order to complete the execution. When representing communication, incoming events are messages to be transmitted, and the resource required for processing is the communication link on which the transmission occurs, described by the transmission time. In either case, it is assumed that incoming events are queued as they arrive. Once processing of an event is finished, a new event is generated and sent to other components. This event represents result of the task computation or delivery of the message on the communication link. Availability of resources to perform the processing is affected by other components sharing the resource.

An abstract component, then, has two types of inputs and outputs: event streams and resource supplies. Characteristics of an event stream are represented as a function $e : R^+ \rightarrow N \times N$, where R^+ is the set of non-negative reals and N is the set of non-negative integers. The function $e(\Delta)$ gives upper and lower bounds on the number of events in any interval of time of duration Δ. Similarly, resource supply is represented as a function $r : R^+ \rightarrow R^+ \times R^+$, giving a lower and an upper bounds on the amount of resource available to the component in any interval of duration Δ. We refer to functions e and r as event arrival and resource curves, respectively. Note that each function contains both the lower and upper bound curves. When we need to refer to one of the two bounds or an event arrival or resource curve, we add the superscript to indicate this: e^l or r^l for lower bound curves and e^u, r^u for upper bound curves.

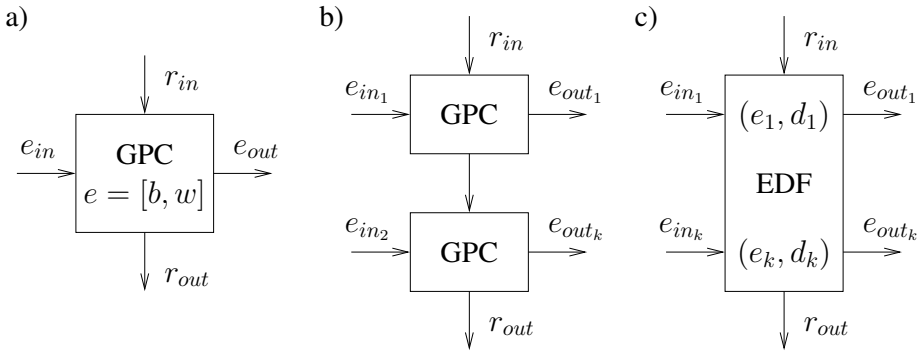


Fig. 2. Abstract processing components

A commonly used way to specify event streams that are input to the whole system is the (p, j, d) model. (p, j, d)-curves are roughly periodic but at the same time are subject to significant bursts in the short term. Here, p is the long-term period of the stream; j is jitter, which characterizes burstiness of the stream, and d is minimal separation between two consecutive events. For a (p, j, d)-curve e, we have

$$e^l(\Delta) = \left\lfloor \frac{\Delta - j}{p} \right\rfloor$$

$$e^u(\Delta) = \min\left(\left\lceil \frac{\Delta - j}{p} \right\rceil, \left\lceil \frac{\Delta}{d} \right\rceil \right)$$

Each abstract component transforms input event arrival and resource curves into respective output curves. A simple example of a component is the generalized processing component (GPC), shown in Figure 2,a. The component represents, for example, a single preemptible task scheduled under a fixed-priority scheduler. The task is characterized by the execution time e, a tuple of real numbers representing best-case and worst-case execution times. Resources that are unused by a task are available to lower-priority tasks. Thus, the output resource curve of a GPC, which characterizes the unused resource after the execution of the GPC, becomes the input resource curve of the component representing the task at the next lower priority level. If all tasks scheduled on the same processor by a fixed priority scheduler have distinct priorities, components representing them can be chained together via their resource curves in the order of decreasing priorities, as shown in Figure 2,b. RTC expressions used to calculate output event and resource curves of a GPC component from its input event and resource curves and the e parameter can be found in [1]. In more complicated cases, a component represents multiple tasks. For example, if earliest deadline first (EDF) scheduling is used, all tasks have to be analyzed together, represented by a component that has one input and one output resource curves, but multiple pairs of input and output event arrival curves corresponding to different tasks, as illustrated

in Figure 2,c. In addition to the execution time, each task specifies a deadline. Calculation of output curves for an EDF component are based on the *demand bound function* technique similar to [11]. A similar component is defined for FIFO scheduling - for example, covering the case of fixed-priority threads with equal priorities as well as non-preemptible transmission of messages on a bus. The deadline is not specified in this case.

There are several auxiliary components that operate on streams. We use three kinds of auxiliary component in our modeling. The first kind, which we visually denote in diagrams as $\oplus$, lets you merge two streams. Given two event arrival curve functions e_1 and e_2, $e_1 \oplus e_2(t) = e_1(t) + e_2(t)$, and similarly for resource curves. The second auxiliary operator allows us to split streams into multiple substreams. Given an event arrival curve function e, we use an operator $\oslash_p$ defined as $\oslash_p e(t) = \lceil p \cdot e(t) \rceil$. Finally, RTC includes a *greedy shaper* component (GSC). A GSC component ensures that its output event stream is bounded from above by a curve e given to it as a parameter. It achieves this by delaying events in the incoming stream. Thus, a GCS component allows us to limit the amount of traffic at the expense of increasing processing delay and buffer requirements. GSC components will be useful for modeling sporadic threads as we discuss below.

Real-time calculus allows us to calculate two important performance measures for a component. One is the maximum delay d_{max}, an upper bound on the latency of processing an event. The other is the maximum buffer space b_{max}, an upper bound on the size of a buffer necessary to avoid losing incoming events. In the case of the GPC, these values are computed as follows [10]:

$$d_{max} \leq \sup_{\lambda \geq 0} \left\{ \inf\{\tau \geq 0 : e^u(\lambda) \leq r^l(\lambda_\tau)\} \right\}$$

$$b_{max} \leq \sup_{\lambda \geq 0} \{e^u(\lambda) - r^l(\lambda)\}$$

Tool support for the RTC analysis. Modeling and analysis described above are supported by the RTC toolbox for Matlab [18], implemented by Lothar Thiele and his collaborators at the Swiss Federal Institute of Technology (ETH) in Zürich. The toolbox provides Matlab functions to create event arrival and service curves, such as `rtcpjd` for a PJD arrival curve, as well as functions that implement abstract components, such as `rtcgpc` for the GPC component. The toolbox is freely available and can be downloaded from the project web site, along with extensive tutorial for its use.

Abstract components are processed by the toolbox one by one. Given the component and the input event arrival and service curves, output arrival and service curves are computed. Since the input curves may be produced by another component in the model, abstract components need to be processed in the topological order of dependencies between the components. If the graph of dependencies contains cycles, a fixed point needs to be computed, which may require several iterations of abstract component processing. In order to compute the fixed point, cycles in the dependency graph are broken until the graph is

acyclic. Let a curve e be produced by a component c_1 and used as input by a component c_2. If e is used to break a cycle, the input to c_2 is initialized with a PJD curve $e_0 = (0, 0, 0)$. After one iteration of component processing, an output curve e_1 is produced and used as input to c_2 for the next iteration. The process is repeated until the computed curve e_{i+1} is equal to the input curve e_i. The fixed point is computed iteratively in a Matlab script.

3 Translating AADL to RTC

3.1 Properties Used in the Translation

The AADL model should contain enough information to extract parameters necessary to populate the RTC model. These parameters, primarily, describe duration of individual processing or communication steps as well as input event arrival and resource curves.

Processor components. Every processor that has thread components bound to it should have the `Scheduling_Protocol` property set. Supported scheduling protocols are RMS (rate-monotonic scheduling), EDF (earliest deadline first), and FPS (fixed priority scheduling with explicitly assigned priorities).

Thread components. Every thread component in the AADL model should specify the property `Dispatch_Protocol`. Allowed values of this property are `periodic`, `aperiodic`, and `sporadic`. If the dispatch protocol is periodic or sporadic, the property `Period` needs to be specified. Thread execution time needs to be specified using the property `Compute_Execution_Time`. The property specifies an interval, $[b, w]$, $b \leq w$, where b is the best-case execution time and w is the worst-case execution time.

If the thread is bound to a processor with the FPS scheduling policy, the thread should have the `Priority` property specified. If the thread is bound to a processor with the RMS scheduling policy, the thread should be periodic or sporadic. In this case, threads mapped to the processor have priorities implicitly assigned according to the RMS policy; that is, inversely proportional to the period of the thread. If the thread is bound to a processor with the EDF scheduling policy, it should have the `Deadline` property specified. If the thread is periodic or sporadic, the deadline is, by default, equal to the value given by the `Period` property. A thread cannot have the `Priority` property specified for the case RMS and EDF cases.

Bus components. The time to transmit a message across a bus depends on the bandwidth and propagation delay of the bus, given by the `Bandwidth` and `Propagation_Delay` properties. In addition, the size of the message needs to be obtained from the data type of the connection that is bound to the bus. The data type in AADL is specified by the data component type, which offers the property `Source_Data_Size`.

Ports. Input event and event data ports of the system are the points where flows of messages enter the system. Their properties are used to construct event arrival curves in the RTC model. We use two properties of a data port to capture parameters of the arrival curve. Property `Input_Rate` specifies a range $[r_{min}, r_{max}]$ of time values. We interpret r_{max} as the long-term period of the stream and r_{min} as the minimum separation d between two events in the (p, j, d) stream model. If the jitter in the stream needs to be specified, the property `Input_Jitter` is used. By default, the value of jitter is 0.

If a thread has multiple outgoing ports event, by default an event is produced on every output port at the end of every invocation of the thread. This can lead to an overly pessimistic message traffic. More precise information can be specified using `Output_Rate` property. In its simplest form, the property can specify a number between 0 and 1, representing the fraction of thread outputs that are transmitted through the port. This information can be used to bound event curve and achieve more accurate calculations. More complicated probability distributions can be associated with the `Output_Rate` property, which we do not consider in this work. While RTC analysis techniques do not utilize probabilistic information directly, it appears that such information can still be useful to improve precision of the analysis. In particular, we plan to investigate how these distributions can be used to limit bursts of events on the arrival curve.

3.2 Abstract Component Graph Construction

The first stage in the construction of the RTC model is to extract a graph of dependencies between threads and network messages in the AADL model. Each node in the graph corresponds to a thread or a network message (that is, AADL connection mapped to a bus), and each edge represents an event or resource dependency between them. A precise definition of the graph is given below. Intuitively, this dependency graph is very close to the graph of abstract components, except that some abstract components may contain multiple threads (e.g. the EDF component, see Section 2.2). In the second stage, we collapse some of the nodes in the graph together to form abstract components. For the remainder of this section, the term *node* refers to a node in the dependency graph.

The graph of dependencies, which we denote $\mathcal{D}$, has its set of nodes partitions into the following sets of nodes: 1) computation nodes n^c; 2) message nodes n_m; 3) event source nodes n^e; and 4) resource nodes n^r. The set n^c contains the node n^c_t for every thread t in the AADL model. The set n^m contains the node n^m_c for every semantic connection c in the AADL model that is bound to a bus. The set n^r contains the node n^r_p (respectively, n^r_b) for each processor p or bus b component in the AADL model. Finally, the set n^e contains the node n^e_{pt} for each input event or event data port pt at the top level of the AADL model and one node n^e_t for each periodic thread t, which represents invocation of the t by the system timer.

Further, $\mathcal{D}$ has the set of directed edges, partitioned into event edges $\rightarrow_e$ and resource edges $\rightarrow_r$. Event edges represent the flows of events through the system, according to the following rules:

- For each periodic thread t, there is an edge $n_t^e \rightarrow_e n_t^c$;
- For each semantic event connection c with the ultimate source t and ultimate destination t', there is an edge $n_t^c \rightarrow_e n_{t'}^c$ if c is not bound to a bus and two edges $n_t^c \rightarrow_e n_c^m$ and $n_c^m \rightarrow_e n_{t'}^c$, otherwise;
- Finally, for each port pt and each thread t that can be reached from pt by traversing a chain of entry connections, there is an edge $n_{pt}^e \rightarrow_e n_t^c$.

Resource edges are added to represent resource supply. Let $t_1, t_2, \ldots, t_k$ be a sequence of threads mapped to a processor p, such that the order of the sequence is consistent with the decreasing order of priorities of the threads. That is, if $i < j$, the priority of t_i is no less than the priority of t_j. If p has the EDF scheduling policy, any order is acceptable. Then, $\mathcal{D}$ contains resource edges $n_p^r \rightarrow_r n_{t_1}^c$, $n_{t_1}^c \rightarrow_r n_{t_2}^c$, $n_{t_2}^c \rightarrow_r, \ldots, \rightarrow_r n_{t_k}^c$. For a bus component and connections bound to it, a chain of resource nodes is constructed in a similar way.

Once $\mathcal{D}$ is constructed, we transform it into a graph of abstract components by adding several auxiliary nodes as described below and by merging the nodes that have to be processed together. Three kinds of auxiliary nodes are introduced:

- if a node n has multiple incoming event edges, all of them are redirected to a new merge node n_+ and a new edge $n_+ \rightarrow_e n$ is added (see Figure 4);
- if a thread t has an output port with the output rate less than one, a scaling node is added to the respective outgoing edges of the thread node. A scaling node with factor $0 < q < 1$ transforms an event arrival curve by multiplying both lower and upper bounds of the curve by q;
- a GSC node is added to the incoming event edge of a node corresponding to a sporadic thread. We discuss this in more detail below.

Let $n_{t_i}^c, n_{t_{i+1}}^c, \ldots, n_{t_{i+j}}^c$ be the nodes corresponding to threads bound to the same processor, or connections bound to the same bus, which have equal priorities. In the case of an EDF processor, these would be all threads bound to the processor. All of these nodes are merged into a new node and any edge incident to any of these nodes is now incident to the new node.

Once the graph of abstract components is constructed, the RTC model can be generated in the format accepted by the RTC Matlab toolbox. First, if the graph contains cycles, the need to be broken to enable fixed point computation. Graph edges that close cycles are identified by depth-first search of the abstract component graph. Whenever a backward edge is found, it is replaced with auxiliary input stream necessary for the fixed point computation as described at the end of Section 2.2. Then, the resulting acyclic graph is topologically sorted, and each node is translated into an abstract component in the format of the RTC Matlab toolbox. Nodes that correspond to processors with EDF policies are turned into the EDF abstract components; merged fixed-priority nodes with equal priorities are turned into FIFO abstract components; finally, nodes that were not merged appear as GPC components.

Parameters of the abstract components are taken from AADL properties. For a component that corresponds to a thread or a group of threads, parameters are obtained from the `Compute_Execution_Time` and, in the case of EDF components, `Deadline` properties. For a component that corresponds to a bus message

or a group of messages, parameters are computed from the `Propagation_Delay` and `Bandwidth` properties of the connection, and the size of the message that is given by the data type of the incident ports. Given the message with the size of s bytes, which is transmitted over the connection with delay d and bandwidth b, the resource requirement for the respective abstract component is computed as $b * s + d$.

Example. Consider again the example in Figure 1. Assume that *proc1* is using the FPS policy and *proc2* is using EDF. Further, let *t11* have a higher priority than *t12*, and that messages from *s1* have higher priority than messages from *s2*. Finally, assume that the output rate for port *p_out1* is 0.8 and for port *p_out2* it is 0.2. The graph $\mathcal{D}$ for this example is shown in Figure 3. Next, we add auxiliary nodes to the dependency graph. Both messages from the network and externally arriving events cause the dispatch of the thread *t21*, therefore the arrival curves of the two streams are added together. Also, the output event stream of thread *t22*

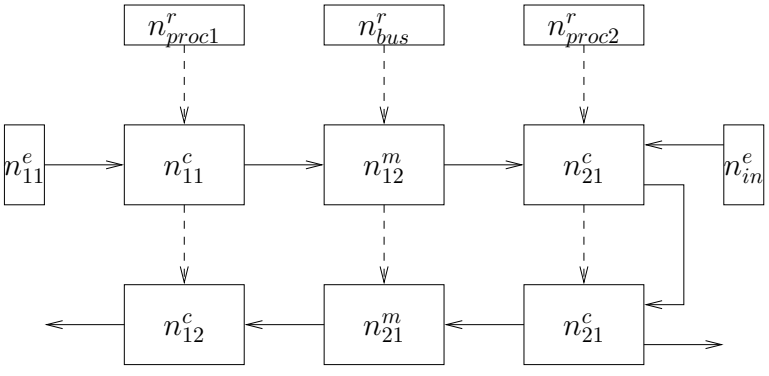


Fig. 3. Graph of dependencies between threads and messages

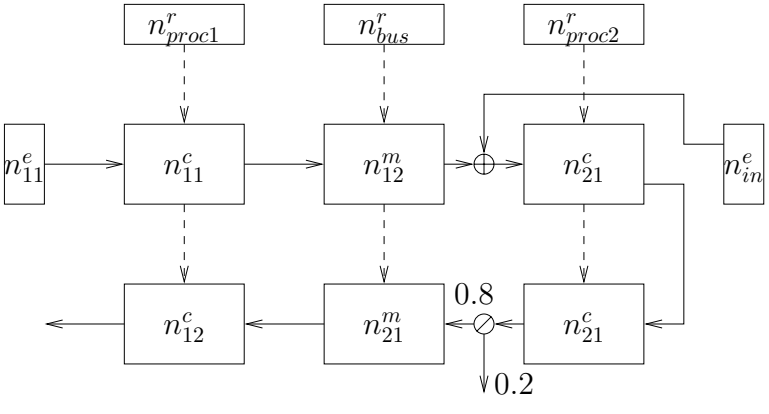


Fig. 4. Auxiliary merge and scale nodes in the dependency graph

is scaled differently according to the `Output_Rate` properties of its output ports. The resulting graph is shown in Figure 4. Note that we abuse the notation for the operator $\oslash_p$ for visual convenience, and show scaling factors on the outgoing edges of the single scaling node. Finally, since threads on the processor *proc2* are scheduled according to EDF, they need to be put together into the same abstract component. Note that the event edge from *t21* to *t22* becomes a self-loop, which would require us to iterate the analysis in order to compute the fixed point. The graph of abstract components is shown in Figure 5.

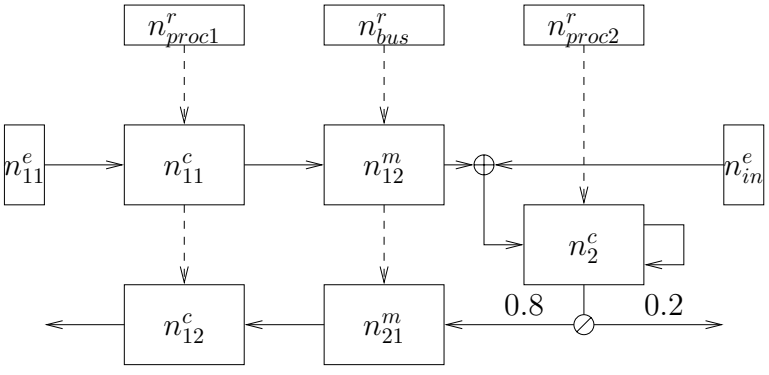


Fig. 5. Graph of abstract components

Sporadic threads. Sporadic threads have the `Period` property that specified the minimum separation between incoming events that cause the dispatch of the thread. The AADL standard specifies that if events arrive more often, they are queued until minimum separation is achieved. This is exactly the behavior that GSC components offer. If the value of the thread's period is p, we create a PJD curve $(p, 0, 0)^u$ – that is, the worst-case curve for a perfectly periodic arrival of events. This curve is used as the parameter of the GSC component. By placing this GSC component in front of the GPC component representing the thread, we achieve the additional buffering that differentiates sporadic from aperiodic threads.

Data connections. Unlike event connections considered above, flows along AADL data connections do not cause thread dispatches. Therefore, these data flows typically do not appear as input event streams to RTC abstract components. However, data connections that are bound to buses introduce additional traffic on the bus and contribute to the delay in transmitting of other messages on the same bus. Thus, a data connection d from thread t to thread t' yields an event message node n^m_d in $\mathcal{D}$, in exactly the same was as message nodes for event connections. We also introduce an event edge from the computation node n^c_t of the source thread node to the message node n^m_d. Because this is a data connection that does not cause dispatch, there is no event edge from n^m_d to the destination thread node $n^c_{t'}$.

4 Case Study

In order to evaluate the proposed transformation and scalability of the analysis, we conducted a case study of a wireless sensor network architecture based on the application level ISA100 proposal [8].

4.1 Overview

The case study represents a collection of sensor nodes connected by a multihop backbone to a gateway, which is in turn connected to a wired network that includes operator nodes, alarm handlers, history loggers, etc. The architecture of the system is informally represented in Figure 6. We do not model the wireless network explicitly; however, it affects the wireless subsystem in two ways. On the one hand, the wired network provides a load to the wireless subsystem, which comes in the form of a stream of operator requests. These requests are passed by the gateway to the wireless subsystem. On the other hand, other kinds of load are assumed to directly affect only the gateway. These loads can have widely varying characteristics, from firmware downloads, which are infrequent transmissions of large size on the one end of the spectrum; to frequent bursts of short requests that are handled by the gateway - for example, ARP broadcast messages - on the other end of the spectrum. We refer to the latter kind of load on the wired network as network noise. Although these additional loads from the wired network are handled by the gateway, they can affect the wireless subsystem when it comes to handling flows of messages from sensor nodes. Messages from sensor nodes need to be transmitted across the wired network. If the wired network is busy, these messages need to be stored in the gateway, delaying their processing and increasing the buffer space requirements in the gateway.

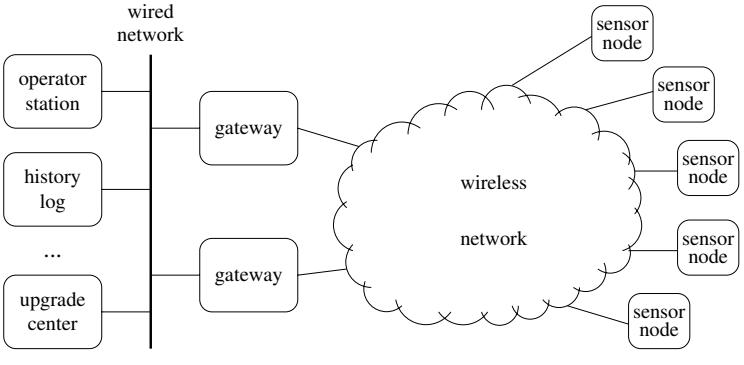


Fig. 6. System architecture for the case study

Data communication with sensor nodes. Sensor nodes support three modes of communication. First, sensor data are periodically published using the TDMA mechanism. Second, sensor nodes can transmit their status information (referred to as parameter values) in response to requests from operators. This communication proceeds in the "client-server" mode using a CSMA protocol, which does not guarantee the absence of collisions. Finally, sensor nodes can spontaneously report alarms that indicate abnormal conditions. Alarm handling is described in detail below. Alarm messages compete with client-server messages for access to the medium.

In order to minimize the number client-server messages traversing the wireless network, the gateway uses a cache. When a request for a particular parameter value arrives, it is checked against the cache and, if found, the value is returned immediately. Otherwise, the request is forwarded to a sensor node across the wireless network. The received response is stored in the cache and then returned to the operator node that issued the request.

Alarm handling. The gateway receives alarm messages from sensor nodes and forwards them to alarm handlers across the wired network. In order to cope with bursts of alarms, incoming alarm messages are stored in a FIFO queue. Each alarm message is acknowledged upon being queued to the node that raised the alarm. The stream of alarm acknowledgment messages adds to the CSMA traffic on the wireless network. If the alarm queue becomes full, further incoming alarms are dropped without being acknowledged. The alarming sensor node, in that case, eventually times out and retransmits the alarm.

4.2 Architecture Modeling in AADL

We modeled the system in the graphical AADL notation. Figures in this section are screen captures from the AADL graphical editor developed by the TOP-CASED project [16]. Figure 7 shows the overall architecture of the system with one gateway and one sensor node. We model the TDMA and CSMA parts of the wireless medium as two separate networks[1]. Note that we do not model the nodes on the wired network that serve the sources and destinations of message flows through the system. Instead, we model an open system, where sources and sinks of message streams are represented as input and output ports. This modeling device allows us to represent parameters of input streams as properties of the ports and easily vary them in the architecture evaluation. The port labeled *fault* is another modeling device that allows us to represent spontaneous raising of alarms by sensor nodes and capture parameters of alarm streams.

Figure 8 represents the architecture of a gateway. The assumption used in modeling was that each kind of incoming message is handled by a separate thread. Ports on the left-hand side of the diagram represent communication on

[1] AADL version 2 provides a more natural way of modeling by using *virtual buses*, which would reflect that both parts are the same medium. However, AADL 2 lacks tool support that we needed in this work.

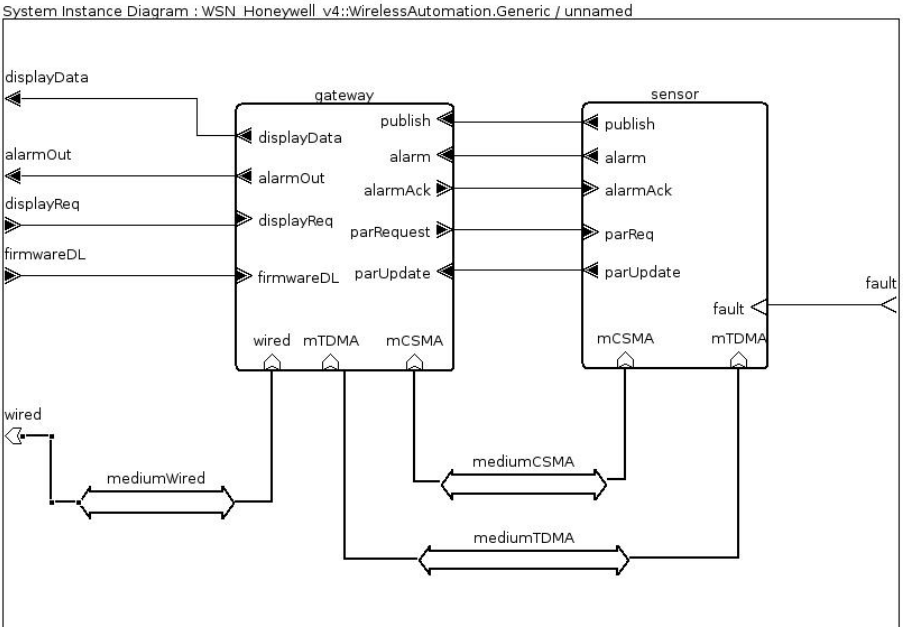


Fig. 7. AADL architecture of the case study

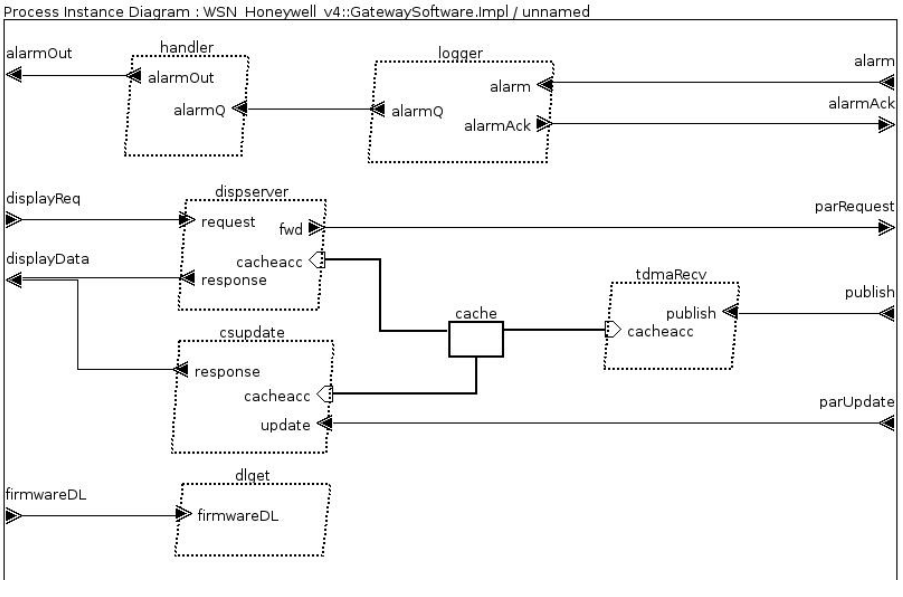


Fig. 8. AADL architecture of the gateway

the wired network, while ports on the right-hand side represent wireless communication.

The top portion of Figure 8 represents the alarm stream. The *logger* thread receives alarm messages from the wireless network, puts them into the alarm queue, and sends acknowledgments back. The *handler* thread takes alarm messages from the queue and transmits them across the wired network. Note that the alarm queue is not represented explicitly. Instead, we utilize the fixed-size FIFO queue of the input data port of the *handler* thread defined by the AADL semantics. Since alarms are dropped when the alarm queue is full, we choose the `Drop_Newest` value for the `Overflow_Handling_Protocol`, which specifies the behavior of the port in the case of the queue overflow.

The middle portion of Figure 8 represents the parameter request stream. The *dispserver* threads accepts operator requests from the wired network, consults the cache and either returns the parameter value or forwards the request through the wireless network. The *csupdate* thread receives the response messages from the wireless network, updates the cache, and transmits the updated value to the operator. Note that the sensor values periodically published by sensor nodes are also stored in the cache and served to operators by the mechanism described above.

Finally, the bottom part of the diagram shows the additional load imposed on the gateway by the wired network in the form of firmware updates or network noise as described above. It is represented by the *dlget* thread that serves as the sink for the flow of these messages.

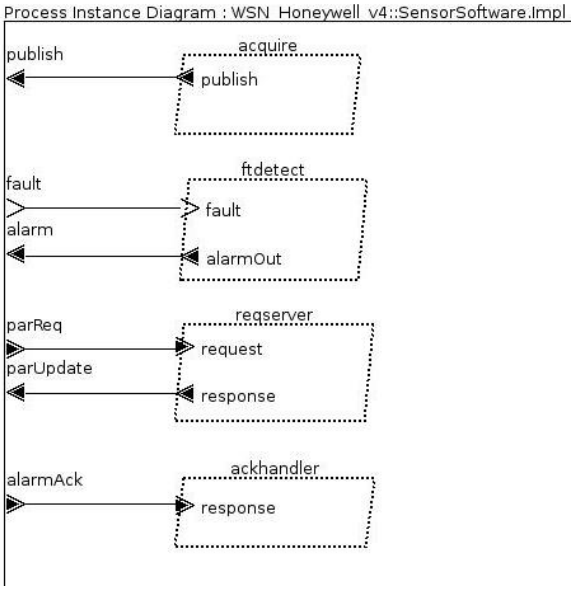


Fig. 9. AADL architecture of the sensor node

All software on the gateway execute on the same processor under fixed-priority scheduling. The *dlget* thread is given the highest priority to maximize the effect of extraneous loads onto the core of the system. The periodic publish thread has the next highest priority, followed by the alarm logger thread, alarm delivery thread, *dispserver* thread, and, finally, *csupdate* thread. This priority assignment lets the client-server communication suffer the most interference from other aspects of the system. Priority assignments can be easily changed at the AADL level and analysis can be repeated to stress other parts of the system such as alarm handling.

Figure 9 shows the architecture of the sensor node. It contains a periodic thread that publishes sensor data, a fault detection thread that transmits alarms, a thread to service parameter requests, and a thread that collects alarm acknowledgments. Here, we also assume that the processor uses fixed-priority scheduling, with the periodic publish thread having the highest priority, followed by the alarm handling thread, client-server thread, and acknowledgment thread, in that order.

4.3 RTC Model for the Case Study

Figure 10 shows the RTC model of the architecture described above, with one gateway and one sensor node. Note that the client-server messages on the wireless network are processed together in one abstract component. All other resources are assumed to use fixed priorities. The event source node *a_pub* represents the

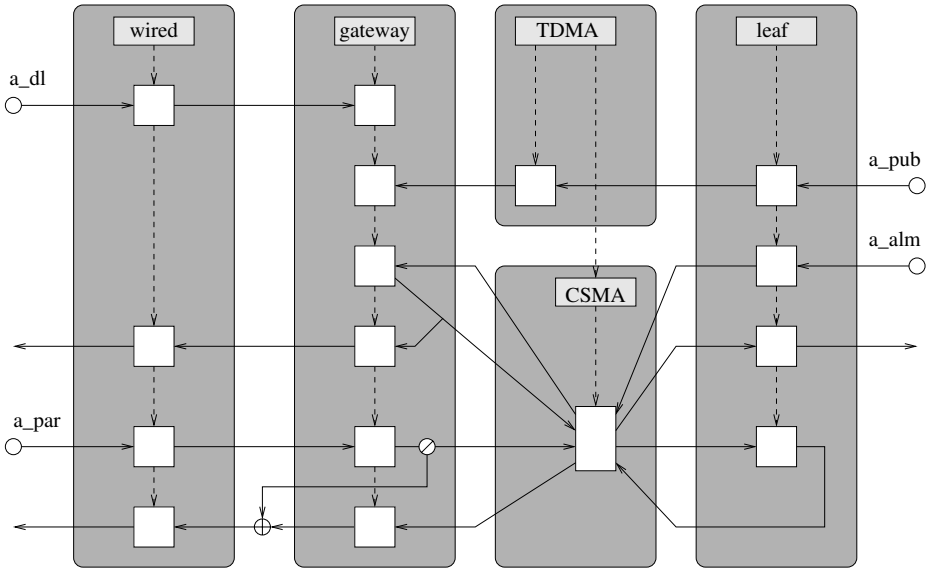


Fig. 10. RTC model of the architecture with one sensor node

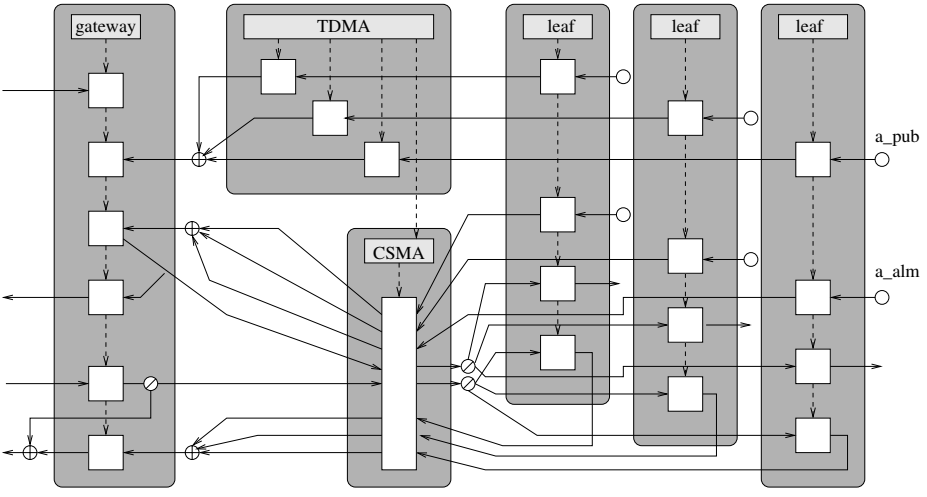


Fig. 11. RTC model of the architecture with three sensor nodes

periodic publishing of sensor readings, while the other three event source nodes correspond to the three input ports of the AADL system in Figure 7.

Figure 11 shows the evolution of the RTC model as more sensor nodes are added. The part of the RTC model that describes the wired network is unchanged compared to Figure 10 and is not shown to avoid cluttering the figure. Event streams from the sensor nodes are merged together before entering the gateway, and event streams from the gateway to the nodes are split and proportionally scaled as they enter the sensor nodes. Each sensor node publishes its readings using a separate TDMA slot, without interference from other nodes. In the RTC model, every TDMA slot is represented as a separate resource. CSMA communication happens in the interval that remains after all TDMA slots have been allocated. This interval is assumed to be contiguous (that is, TDMA slots are allocated next to each other within the service interval. Note that, as more sensor nodes are added, the CSMA interval becomes smaller, affecting performance of client-server communication.

One can notice that the RTC model does not capture the following aspect of the sensor node behavior. If an alarm is not acknowledged by the gateway, the sensor node is supposed to retransmit the alarm. In the initial version of the model, we tried to represent this aspect directly: the acknowledgment traffic was split in some proportion into two flows. One flow, representing acknowledgments, traveled as client-server messages though the network. This flow is present in the current model. The other part represented a virtual flow, capturing the fact that each dropped message causes an invocation of the thread responsible for alarm transmission. We thus viewed dropped messages as the stream of timeouts, which was merged, with the appropriate delay, with the flow of faults that also trigger the same alarm thread. It turns out that this model, which seems more faithful

to the real system, has two drawbacks. First, it is not clear, in which proportion should the acknowledgment traffic be split. It becomes another parameter to be provided by the user, who does not have any principled basis to supply this parameter value. More seriously, the RTC model with such feedback turned out to be hard to analyze: the fixed point computation did not converge in a reasonable number of steps, and the processing time of a step increased dramatically with each next iteration. To avoid this problem, the final version of the model shown above followed a different approach. We assumed that every alarm is acknowledged, so no retransmissions were necessary. We then calculated the buffer requirements for the alarm queue. Once the system satisfies the buffer requirements, no alarms are dropped and the assumption is satisfied.

4.4 Analysis Results

During the analysis, we considered several configurations of the model. The configurations differed in the parameters of the highest-priority load imposed on the system by the wired network. This turned out to be the most significant factor to affect the running time of the analysis of a single model instance (that is, with the number of nodes fixed). Three configurations were explored. Two configurations describe the "firmware download" kind of wired network load. One had a period of 0.5 hours with very high bursts (jitter equal to 2 hours), and a minimum separation of 1 minute. The other configuration has a reduced jitter parameter value equal to 15 minutes with the other parameters unchanged. The third configuration was the "network noise" load with 1.8 second period, 0.6 seconds minimum separation, and 7.2 seconds jitter. Parameters of configurations were chosen to be on the opposite ends of the parameter spectrum and do not directly represent traffic parameters of the real system.

All experiments described below were performed using Matlab version R2007b and the RTC toolbox Version 1.1 beta 1.03. The platform used was a ThinkPad T61 laptop with a 2.0 GHz dual-core processor and 1GB of main memory, running Ubuntu linux.

End-to-end delay calculations. The first set of results describes end-to-end delays for different event streams in the system. As an example, Figure 12 shows end-to-end delay of the alarm stream, measure from the moment an fault is detected by a sensor node to the moment the alarm is delivered to the destination, an alarm handler node on the wired network. For relatively low network utilization, up to eight sensor nodes, the calculated delay is growing linearly with the number of nodes. However, as Figure 12,b demonstrates, once the network capacity is exceeded, the delay grows up dramatically.

Alarm queue. Figure 13 shows the buffer requirements for the alarm queue as a function of the number of nodes. In this case, also, we can see that buffer requirements initially increase linearly with the number of sensor nodes and then, upon reaching a threshold suffer a sharp increase that indicates that the network capacity needs to be enhanced.

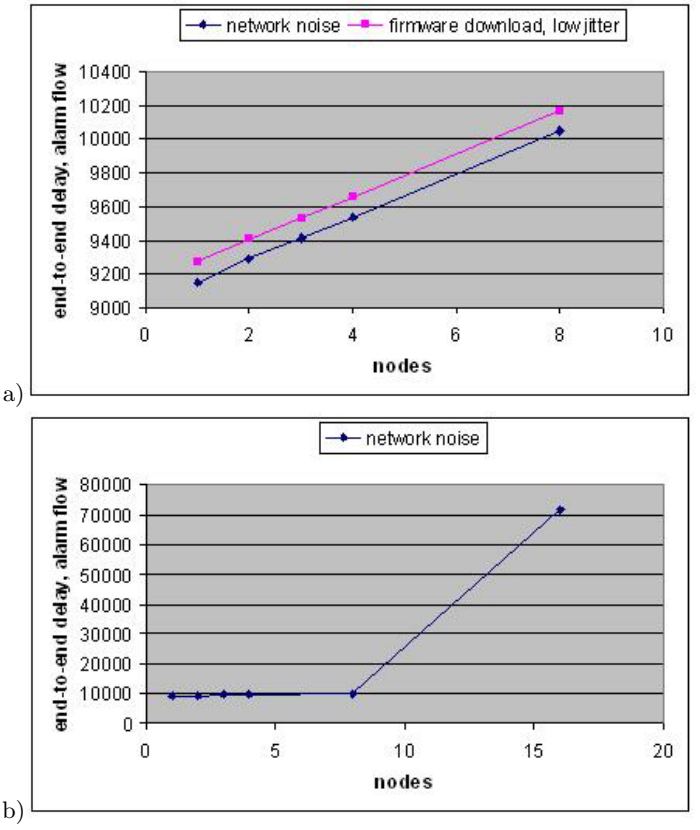


Fig. 12. End-to-end delay of the alarm stream

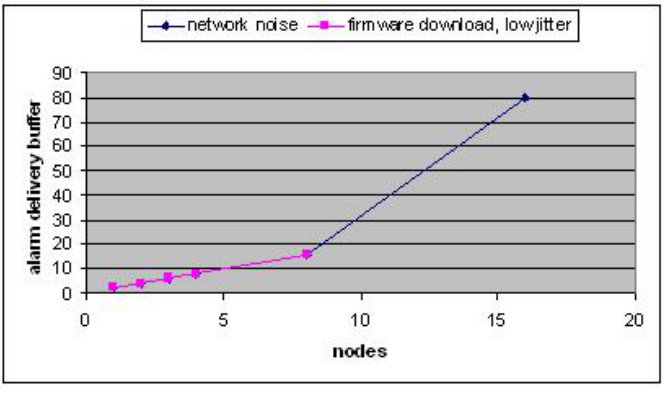


Fig. 13. Required alarm queue size

Scalability. Figure 14 gives the total running times of the experiments as reported by the RTC toolbox. Clearly, the running time is superlinear with respect to the number of nodes. Note that not only the total running time of each experiment increases with the range of timing constants in different configurations, but also the rate of increase (slope of the curve) depends on the range as well. Note, however, that larger numbers of nodes required more fixed point iterations to complete the analysis: from four iterations for up to four nodes to six iterations for sixteen nodes. To account for the increased number of iterations, we also calculated analysis time per one iteration, which is shown in Figure 15. For comparison, the total time for the network noise configuration is also shown in Figure 15. Time per iteration can be seen to grow much slower than the total time.

The obvious conclusion from the data is that RTC-based analysis is sensitive to the range of time constants. In all cases, the smallest time constant was on

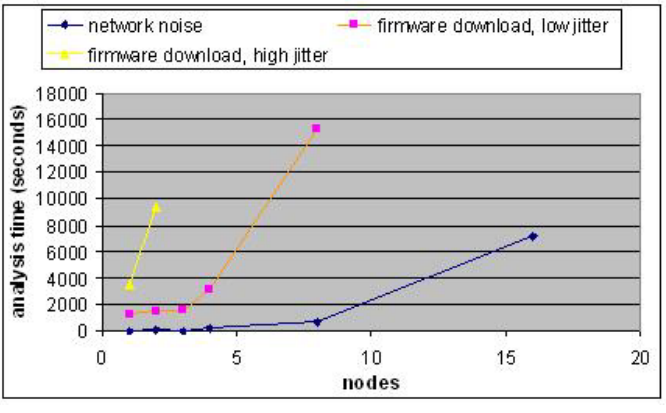


Fig. 14. Total running time of experiments

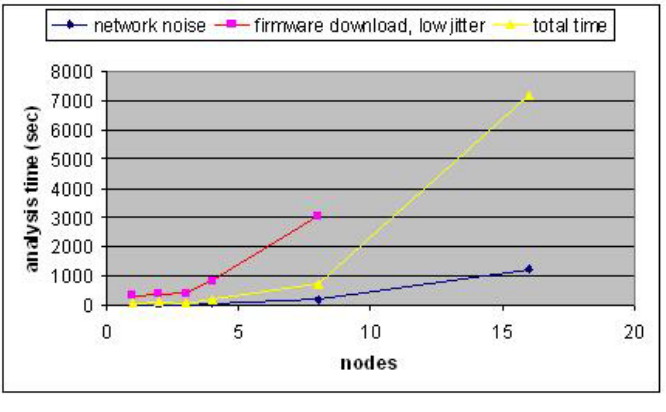


Fig. 15. Running time per iteration

the range of 1 ms. The bursty firmware download configuration was by far the most time-consuming configuration to analyze. It had the jitter parameter value as the largest time constant in the model, and reducing just this value in the second firmware download configuration improved analysis time dramatically. Further reducing timing parameters of that event stream in the network noise configuration improved analysis time further.

5 Conclusions and Future Work

We consider analysis of timing and performance properties of systems expressed in the architecture description language AADL. We presented an algorithm to extract from such an architectural model an analytical model based on Real-Time Calculus, and discussed properties that can be determined using this model. We applied this analysis technique to a case study based on a wireless sensor network architecture. The case study included modeling of a typical architecture and analysis of several variants of the model different number of network nodes and workload parameters and comparative analysis of these configurations.

The case study identified two areas, where this modeling and analysis approach requires improvement before it can be applied to real industrial-scale systems. One deficiency is scalability. Current tools allow analysis of relatively small-scale systems. On the one hand, existing tools can be substantially improved with a more efficient implementation using new data structures for event curve representation. On the other hand, research is needed into improved algorithms that would reduce the dependency of running time on the range of timing constants. In the current implementation of RTC, an attempt to combine very small timing constants (such as the millisecond-level scale of message transmission) with very large ones (such as the minute-level of scale of periodic sensor updates) results in a very space-inefficient representation of arrival curves, which in turn adversely affects analysis running time. The other area that needs improvement is the precision of analysis. Results may be excessively conservative, because arrival curves do not reflect any temporal aspects of flow propagation. Precision can be improved by incorporating existing techniques for considering workload variability and event correlations, for example, based on event count automata [2,12].

References

1. Chakraborty, S., Künzli, S., Thiele, L.: A general framework for analysing system properties in platform-based embedded system designs. In: IEEE Design Automation and Test in Europe (DATE) (March 2003)
2. Chakraborty, S., Phan, L., Thiagarajan, P.: Event count automata: A state-based model for stream processing systems. In: Proceedings of the 26^{th} IEEE Real-Time Systems Symposium (RTSS 2005), December 2005. pp. 87–98 (2005)
3. de Niz, D., Feiler, P.: On resource allocation in architectural models. In: Proceedings of the 2008 11^{th} IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC 2008), May 2008, pp. 291–297 (2008)

4. Feiler, P., Gluch, D., Hudak, J., Lewis, B.: Embedded architecture analysis using SAE AADL. Technical Report CMU/SEI-2004-TN-005, Software Engineering Institute (June 2004)
5. Feiler, P., Lewis, B., Vestal, S.: The SAE AADL standard: A basis for model-based architecture-driven embedded systems engineering. In: Workshop on Model-Driven Embedded Systems (May 2003)
6. Feiler, P., Lewis, B., Vestal, S.: The SAE Architecture Analysis & Design Language (AADL) a standard for engineering performance critical systems. In: IEEE International Symposium on Computer-Aided Control Systems Design (October 2006)
7. Hassl, D.: Advanced concepts in fault tree analysis. In: Proceedings of System Safety Symposium (June 1965)
8. ISA100 Wireless Working Group. Draft standard ISA100.11a. Internal working draft (May 2008)
9. Jahier, E., Halbwachs, N., Raymond, P., Nicollin, X., Lesens, D.: Virtual execution of AADL models via a translation into synchronous programs. In: Proceedings of the 7^{th} International Conference on Embedded software (EMSOFT 2007), October 2007, pp. 134–143 (2007)
10. Le Boudec, J.-Y., Thiran, P.: Network Calculus - A Theory of Deterministic Queuing Systems for the Internet. In: Thiran, P., Le Boudec, J.-Y. (eds.) Network Calculus - A Theory of Deterministic Queuing Systems for the Internet. LNCS, vol. 2050, p. 3. Springer, Heidelberg (2001)
11. Lipari, G., Baruah, S.: Efficient scheduling of real-time multi-task applications in dynamic systems. In: Proc. of IEEE Real-Time Technology and Applications Symposium, pp. 166–175 (May 2000)
12. Phan, L., Chakraborty, S., Thiagarajan, P., Thiele, L.: Composing functional and state-based performance models for analyzing heterogeneous real-time systems. In: Proceedings of the 28^{th} IEEE Real-Time Systems Symposium (RTSS 2007), December 2007, pp. 343–352 (2007)
13. SAE International. Architecture Analysis and Design Language (AADL), AS 5506 (November 2004)
14. Sokolsky, O., Lee, I., Clarke, D.: Schedulability analysis of AADL models. In: Workshop on Parallel and Distributed Real-Time Systems (April 2006)
15. Thiele, L., Chakraborty, S., Naedele, M.: Real-time calculus for scheduling hard real-time systems. In: IEEE International Symposium on Circuits and Systems (ISCAS), vol. 4, pp. 101–104 (2000)
16. The open-source toolkit for critical systems (2008), http://www.topcased.org
17. Wandeler, E.: Modular Performance Analysis and Interface-Based Design for Embedded Real-Time Systems. PhD thesis, Swiss Federal Insitute of Technology (2006)
18. Wandeler, E., Thiele, L.: Real-Time Calculus (RTC) Toolbox (2006), http://www.mpa.ethz.ch/Rtctoolbox
19. Wandeler, E., Thiele, L., Verhoef, M., Lieverse, P.: System architecture evaluation using modular performance analysis: a case study. Software Tools for Technology Transfer 8(6), 649–667 (2006)

On Software Certification: We Need Product-Focused Approaches*

Alan Wassyng, Tom Maibaum, and Mark Lawford

Software Quality Research Laboratory
Department of Computing and Software
McMaster University, Hamilton, Canada L8S 4K1
wassyng@mcmaster.ca, tom@maibaum.org, lawford@mcmaster.ca

Abstract. In this paper we begin by examining the "certification" of a consumer product, a baby walker, that is *product-focused*, i.e., the certification process requires the performance of precisely defined tests on the product with measurable outcomes. We then review current practices in software certification and contrast the software regime's process-oriented approach to certification with the product-oriented approach typically used in other engineering disciplines. We make the case that *product-focused* certification is required to produce reliable software intensive systems. These techniques will have to be domain and even product specific to succeed.

1 Introduction

This paper deals briefly with the current state of software certification, why it is generally ill-conceived and some reasons for how (and why) we landed in this mess, and suggestions for improving the situation.

2 Motivation

A certification story: Let us start the discussion with an item that has little to do with software, but is typical of engineered artifacts - a baby walker. Consider a typical baby walker, as shown in Figure 1.

In recent years, there has been considerable concern regarding the safety and effectiveness of baby walkers. In reaction to this concern, we can now consider a certification process we may wish to advocate in order that we may regulate the sale of particular baby walkers. So, what should be the overall thrust of such a certification process? Well, humbly we may suggest that we model the process on certification processes that are common in our domain (software). What may such a process look like? Perhaps something like the list shown below:

1. Evaluate manufacturer's development process.
2. Evaluate list of materials used in manufacture of baby walker.

* Supported by the Natural Sciences and Engineering Research Council of Canada.

C. Choppy and O. Sokolsky (Eds.): Monterey Workshop 2008, LNCS 6028, pp. 250–274, 2010.

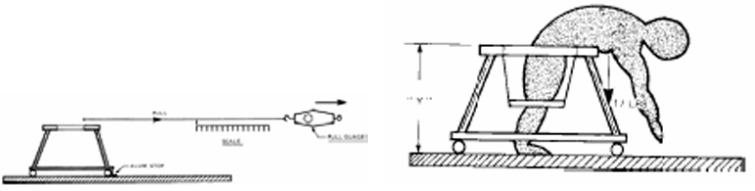
Fig. 1. A Typical Baby Walker

3. Evaluate manufacturer's test plan.
4. Evaluate manufacturer's test results.

Additionally, let us imagine what the manufacturer's submission to the regulators may contain:

1. Process.
   (a) Requirements
   (b) Requirements review
   (c) Design
   (d) Design review
   (e) Manufacturing process
   (f) Manufacturing process review
2. Materials.
   (a) List of materials for each design part
   (b) Safety analysis for each material used
3. Test plan.
4. Test results.

Perhaps a little more detail regarding the testing is required. The manufacturer decided to test two main problems. The first problem was related to quality of manufacture. In this regard, a number of tests were planned and performed regarding the uniformity of the production line and the degree to which the resulting baby walkers complied with the specified design. The second problem related to the tendency of the baby walker to tip. In this regard, two tests of stability were executed. The first test, shown in Figure 2(a), records the force required to tip the baby walker when it is stopped at an abutment. The second test, Figure 2(b), records the moment required to tip the baby walker - simulating a child leaning over.



(a) Stability Test 1 (b) Stability Test 2

Fig. 2. Stability Tests from ASTM Standard F977-00 [1]

So, what may we observe from this example? Our observations may include:

1. It is extremely unlikely that regulators of baby walkers are going to evaluate the manufacturer's development process.
2. The regulator may evaluate the manufacturer's tests, but will definitely run tests independently.
3. The regulator will examine the materials used in the baby walker and determine if they expose the baby to potential harm.
4. **Important:** The regulator is likely to publish and use tests specifically designed to test baby walkers.
   (a) For this example, a number of countries published very specific requirements for baby walkers. For example, *United States Standard ASTM F 977-00 - Standard Consumer Safety Specification for Infant Walkers* [1]. The tests mentioned in Figures 2(a), and 2(b) are not nearly sufficient. A number of dynamic tests have been added to those static tests. The static tests would have been completely useless in determining whether a baby in the walker would fall down unprotected stairs.
   (b) Product-focus is not a panacea either. Canada banned the use of all baby walkers, since Health Canada determined that baby walkers are (probably) not effective, and even product-focused standards may not guarantee safety when the product is ill-conceived [2]. The product-focused standard helped Health Canada arrive at these conclusions, since they could be confident that the products were designed and manufactured well enough to satisfy stated requirements, and so the problems were more fundamental.

> Still, it seems strange to us that to certify a baby walker, a regulator devised a product-based standard and tests baby walkers to that standard, whereas, to certify a pacemaker (for example), regulators use generic software process-oriented standards and regulations!

Government oversight: The easiest software certification to motivate is where the government mandates licensing/certification anyway. In this case, we want to make the case to the regulators/certification authorities that product-focused

certification will result in much more dependable systems than will process-based certification. We believe that this will improve the objectivity and measurability of evidence, thus improving the evaluation process, and thus making the certification process more predictable for all parties. It should also reduce the post-licensing failure rate and facilitate the identification of the cause. This would certainly, ignoring political issues, induce regulators to adopt a more product-focused approach.

Social expectations: Over the past three or four years, we have seen growing interest in software certification, driven in some cases by the public's dissatisfaction with the frailty of software-based systems. Online banking and trading systems have experienced failures that were widely publicized and have caused wide-spread chaos. Software driven medical devices have killed people. Security breaches in software systems have disrupted peoples' lives. There is no reason that software systems should not be certified as fit-for-use and safe - just as most other products are.

Market advantage: There is also a growing realization by commercial companies that if they can market their software with a warranty, that will give them a tremendous marketing edge. So, as soon as they can manufacture certified software at reasonable cost (and that is the difficulty right now), manufacturers will be driven to consider software certification through normal marketing forces.

Component assurance/qualification: Many industries have to use components manufactured by a myriad of different suppliers. For instance, auto manufacturers manufacture some components themselves and buy others from suppliers. These components are becoming more complex and have to work under stringent timing constraints. Product-focused standards and certification are going to be unavoidable if the components are going to be able to deliver dependable service.

Political considerations: Many software producers find the idea of software regulation anathema: witness the move in various jurisdictions (in the US and an abortive one in the European Union) to lower the liability of software manufacturers from even the abysmal levels in place today.

Governments are woefully ignorant of the dangers represented by the low or non-existent levels of regulation in some industries, such as those producing medical devices, cars and other vehicles, financial services, privacy and confidentiality issues in many information systems, etc.

However, the issue is much too large for us, as a society, to ignore any longer.

3 Current Practice

This section describes approaches to software certification in three different application domains. It presents one of our main hypotheses: current practice in software certification is primarily focused on the process used to develop the

software, and that process-focused certification is never going to give us enough confidence that, if a software product is *certified*, then the product will be effective, safe and reliable.

The domains we are going to discuss are:

- medical systems (in the U.S.);
- security systems (primarily in Europe, Japan and North America);
- nuclear power safety systems (in Canada).

3.1 Medical Systems

As an example, we will consider, briefly, the regulatory requirements for medical systems in the U.S. The U.S. Federal Drug Administration (FDA) is responsible for regulating medical systems in the U.S., and they publish a number of *guidelines* on software *validation*, e.g., [3,4]. The FDA validation approach, as described in the FDA guidance documents falls short on describing objective criteria which the FDA would use to evaluate submissions. The documents do not do a good enough job of describing the artefacts that will be assessed. In particular, the targeted attributes of these artefacts are not mentioned, and approved ways of determining their values are never described. The focus of these documents is on the characteristics of a software development process that is likely to produce high quality software. It shares this approach and concern with almost all certification authorities' requirements (as well as those of standards organisations and approaches based on *maturity*, such as CMMI [5]).

3.2 Security Systems

The Common Criteria (CC) [6] for Information Technology Security Evaluation is an international standard for specifying and evaluating IT security requirements and products, developed as a result of a cooperation between many national security and standards organisations. Compared with the FDA's approach for medical systems, the CC has a more systematic and consistent approach to specifying security requirements and evaluating their implementation. The CC does fall into the trap of prescribing development process standards (ISO/IEC 15408) in detail, but, on the other hand, it does a much better job than the FDA guidelines of being measurement oriented.

One very good idea in the CC is that it defines seven levels of assurance, as shown below.

EAL1: functionally tested
EAL2: structurally tested
EAL3: methodically tested and checked
EAL4: methodically designed, tested and reviewed
EAL5: semiformally designed and tested
EAL6: semiformally verified design and tested
EAL7: formally verified design and tested

It is interesting to note that formal methods are mandated, if only at the highest levels of assurance. Testing occurs at all levels, reinforcing that all certification regimes place a huge emphasis on testing. In keeping with a wide-spread movement in trying to make *software engineering* more of an engineering discipline, we see that CC has introduced the concept of "methodical" processes into their assurance levels. Our only cause for concern in this regard, is that the CC community does not seem to require the process to be both methodical and formal (or semi-formal). We do not agree with this, since formality really relates to the rigour of the documentation. It does not necessarily imply that the process is systematic/methodical.

The taxonomy of the CC describes Security Assurance Requirements (SARs) in terms of action elements for the developer, and for the *content and presentation* of the submitted evaluation evidence for the evaluator. Each evaluator action element corresponds to work units in the Common Evaluation Methodology [7], a companion document, which describes the way in which a product specified using the CC requirements is evaluated. Work units describe the steps that are to be undertaken in evaluating the Target of Evaluation (TOE), the Security Target (security properties of the TOE), and all other intermediate products. If these products pass the evaluation, they are submitted for certification to the certification authority in that country.

There are a number of important principles embedded in this approach: the developer targets an assurance level and produces appropriate evidence that is then evaluated according to pre-determined steps by the certifier; this undoubtedly helps in making the certification process more predictable; and the certification process is designed to accommodate third-party certification.

3.3 Canadian Nuclear Power Safety Systems

While proponents of formal methods have been advocating their use in the development and verification of safety critical software for over two decades [8,9,10], there have been few full industrial applications utilizing rigorous mathematical techniques. This is in part due to industry's perception that formal methods are difficult to use and fail to scale to "real" problems. To address these concerns, a method must supply integrated tool support to automate much of the routine mechanical work required to perform formal specification, design and verification.

There have been some notable industrial and military applications of tool supported formal methods, especially for the analysis of software systems requirements (e.g., [11,12,13,14]). Unfortunately, the formal methods advocates concerned, typically were not given the opportunity to fully integrate their techniques with the overall software engineering process. As a result these applications required at least some reverse engineering of existing requirements documents into the chosen formalism. A potential problem of this scenario is that two *requirements specifications* may result: the original, often informal, specification used by developers; and the formal specification used by verifiers.

An example of this problem occurred in 1988. Canadian regulators were struggling with whether to licence the new nuclear power station at Darlington in Ontario. At issue was the "certification" of the shutdown system - it was the first software controlled shutdown system in Canada. The regulators turned to Dave Parnas for advice, and his advice was to require formal proofs of correctness as well as other forms of verification and validation. The regulators and Ontario Hydro - OH (now Ontario Power Generation - OPG) worked together to agree on an approach to achieve this. Most of the time, OH prepared process documents and interacted with the regulator to obtain an agreement in principle. The correctness "proofs" were eventually delivered to the regulator (more than twenty large binders for the two shutdown systems), and a walkthrough was conducted for each of the shutdown systems [15,16]. The end result was a licence to operate the Darlington Nuclear Generating Station. However, the regulator also mandated a complete redesign of the software to enhance its maintainability.

As a result, OPG, together with Atomic Energy of Canada Limited (AECL), researched and implemented a new safety-critical software development process, and then used that process to produce redesigned versions of the two shutdown systems [17]. As a start, OPG and AECL jointly defined a detailed engineering standard to govern the specification, design and verification of safety-critical software systems. The CANDU Computer Systems Engineering Centre of Excellence Standard for Software Engineering of Safety Critical Software [18] states the following as its first fundamental principle:

> *The required behavior of the software shall be documented using mathematical functions in a notation which has well defined syntax and semantics.*

Not only was the software redesigned along information hiding principles, but new requirements documents were also produced. These requirements are formally described, primarily through the use of tabular expressions (function tables) [19]. In fact, the current implementation of the software engineering process makes extensive use of tool supported tabular expressions [20]. The process results in the production of a coherent set of documents that allows for limited static analysis of properties of the requirements. The process also includes a mathematical verification of the design, described in the Software Design Description (SDD), against the software requirements documented in the Software Requirements Specification (SRS). This project is then an example of one in which, from the start, the software development process itself was designed to use formal methods and associated tools to deliver evidence for the licensing (certification) of the resulting system.

A model of the process, including integrated tool support, that was applied to the Darlington Nuclear Generating Station Shutdown System One (SDS1) Trip Computer Software Redesign Project is shown in Figure 3.

The Darlington Shutdown Systems Redesign Project represents one of the first times that a production industrial software engineering process was designed, successfully, with the application of tool supported formal methods to specification and verification as a primary goal. As we have seen, this was

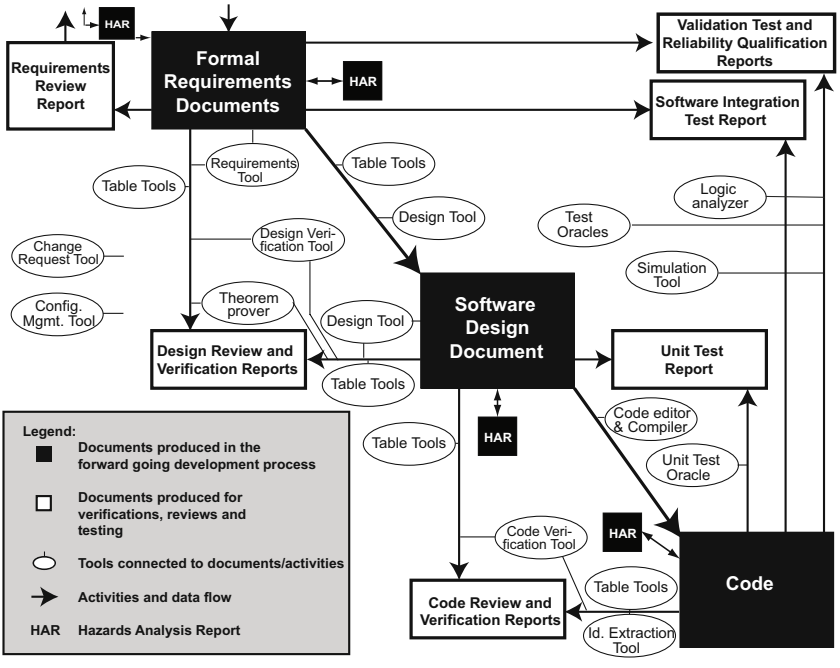


Fig. 3. SDS1 lifecycle phases, documents and tools

necessitated by regulatory requirements, a situation that is becoming increasingly common for industries utilizing software in safety-critical applications. The major factors considered in choosing the particular formal methods for the Redesign Project were: (i) learning curve and ease of use and understanding of the formal specifications, (ii) ability to provide tool support, and (iii) previous history indicating the ability to successfully scale to industrial applications. We now address these three points in more detail.

Since tables are frequently used in many settings and provide important information visually, they are easily understood by domain experts, developers, testers, reviewers and verifiers. From the original Darlington licensing experience, and a trial example of the same verification procedure applied to a smaller scale Digital Trip Meter System [21], OPG had strong evidence that a verification procedure using tabular methods would meet the requirements of the Redesign Project. OPG's confidence in the use of tabular expressions was re-affirmed by domain experts working on SDS1 being able to read and understand the formal requirements specifications, documented almost exclusively by tabular expressions. Also, tabular expressions provide a mathematically precise notation with a formal semantics [19]. Other methods such as VDM or Z utilize unfamiliar notation and special languages with a significant learning curve [22]. The OPG Systematic Design Verification (SDV) procedure avoids this problem through the use of tabular notation in both the requirements and design documents

utilized by all project team members. To create the tabular specifications, custom "light-weight" formal methods tools (in the sense of [23,24]) are used to help create and debug the tables from within a standard word processor. To perform the verification these tools then extract the tables from the documents and generate input files for SRI's Prototype Verification System (PVS) automated proof assistant [25].

Tabular methods are well suited to the documentation of the Shutdown System's control functions that typically partition the input domain into discrete modes or operating regions. Some of the other major benefits of this, and other, tool supported formal methods, include:

- Independent checks which are unaffected by the verifier's expectations,
- Domain coverage through the use of tools that can often be used to check *all* input cases – something that is not always possible or practical with testing,
- Detection of implicit assumptions and ambiguous/inconsistent specifications,
- Additional capabilities such as the generation of counter-examples for debugging, type checking, verifying whole classes of systems, etc.

The creation of the specialized tools that allowed verification to be done with the help of PVS played a large role in making the methods feasible for the larger Redesign Project. A further reason for the adoption of tabular methods is that they have been successfully applied to a wide variety of applications. In particular, they have been used successfully with PVS on problems such as the verification of hardware division algorithms similar to the one that caused the Pentium floating point bug [26].

There are some important points to note about the licensing of the redesigned Darlington Shutdown Systems. Compared with the licensing process for the original system, the redesign licensing process progressed remarkably smoothly. A major contributing factor was that the manufacturer (OPG) had asked the regulator to comment ahead of time on the deliverables for the licensing process. It is true that the regulator wanted to understand (and comment on) the software development process that was to be used in the project. However, the regulator's primary role was to evaluate the agreed upon set of deliverables. The evaluation was more in the form of an audit, in that *post factum*, the regulator specified a slice through the system for which a guided walkthrough was held. The regulator also reviewed major project documents.

3.4 Software Engineers Get It Wrong Again!

The aim of certification is to ascertain whether the product, for which a certificate is being sought, has appropriate characteristics. Certification should be a measurement based activity, in which an objective assessment of a product is made in terms of the values of measurable attributes of the product, using an agreed upon objective function.

Given the choice between focusing on process or product as a means of assessing whether software intensive systems possess the appropriate characteristics,

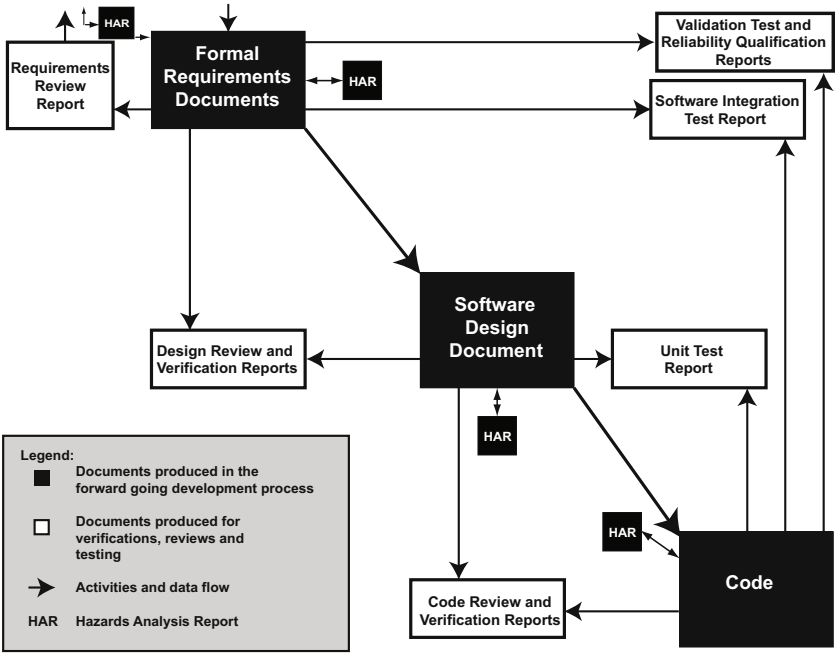


Fig. 4. Idealized software development process

Software Engineers have again made the wrong choice. Classical engineers invariably use product-focused assessment and measurement in evaluating manufactured products. Software products are typically evaluated using process-focused standards. This is tantamount to trying someone on a murder charge - based solely on circumstantial evidence! The process based guarantee is a statistical one over all products, not a guarantee of an individual product.

The focus on CMM (and now CMMI) and other process-oriented standards was (perhaps) necessary to force companies to begin adopting proper engineering methods, but CMMI, as an example, has not progressed to the point where it achieves this.

We are advocating a *product-focused* approach to software certification - we are not saying, however, that software certification regimes should completely ignore the software process. We believe we will always need some notion of an *idealized software development process* in the software certification process. The idea is similar to Parnas and Clement's exhortation to "fake it" [27], in that there has to be agreement on mandatory documents produced during the software development process. For example, a simplified version of the SDS1 development process (Figure 3), could describe a mandated idealized process (see Figure 4 for example), and the certifiers could then evaluate product evidence such as documents and the application itself, without any consideration given to the quality of the development process that actually was followed.

4 Evaluating Process Is Easier

An obvious question arises: "Why did we (software engineers) turn to evaluating process rather than evaluating the final product(s) directly"? The answer, as usual in multi modal disasters, is complicated.

Evaluating the software development process is much easier than evaluating the software product itself.

- We have no real consensus on absolutely essential metrics for products.
- Ironically, even if we did have consensus on essential metrics, what metrics would help us evaluate the dependability of software products directly?
- It is widely accepted that testing software products completely is not possible. One of the major differences between software products and more traditional, physical products, is that the principle of continuity does not apply to software products. Since software engineers felt that even a huge number of test cases could not guarantee the quality of the product, we turned to supportive evidence, hoping that layers of evidence will add up to more tangible *proof* of quality/dependability.

Other disciplines introduced an emphasis on process. From general manufacturing to auditing, the world started putting more and more emphasis on process. We should be clear - there is a huge difference between the manufacture of a product and the certification of that product. We need good manufacturing processes and we also need effective certification processes. We have no hesitation in agreeing that a company needs a good software development process. When we discuss the difference between *process-focus* and *product-focus*, we are really looking at where the certification process should place its emphasis. The world-wide emphasis on manufacturing process made it easy for software certifiers/regulators to concentrate on evaluating a manufacturer's software development process and thus appear to be achieving something worthwhile in terms of certifying the manufacturer's products. We think that certification standards like CMMI and ISO 9000 tell us about the care and competence with which a company manufactures its products. It tells us very little directly about a specific product manufactured by the company.

5 Engineering Methods

We want to make the distinction between a proper engineering method, on the one hand, and having a well defined process as usually understood in software engineering, on the other hand. We begin by discussing the nature of engineering as a discipline. Over the years, engineering has been defined in a number of ways. A useful definition of engineering is the one used by the American Engineers' Council for Professional Development (AECPD). It defines Engineering as: "The creative application of scientific principles to design or develop structures, machines, apparatus, or manufacturing processes, or works utilizing them singly or in combination; or to construct or operate the same with full cognizance of

their design; or to forecast their behavior under specific operating conditions; all as respects an intended function, economics of operation and safety to life and property." Within this context, we need to consider what it means to use "engineering methods". A classic description of the Engineering Method was presented by B.V. Koen [28] in 1985. Koen presents the view that the Engineering Method is "the use of engineering heuristics to cause the best change in a poorly understood situation within the available resources". When one reads the explanation of this, it becomes clear that Koen is talking about a systematic process, that uses a set of *state-of-the-art* heuristics to solve problems. He also makes the point that the state-of-the-art is time-dependent. Before we discuss software engineering in particular, we lay the foundation by exploring concepts in the epistemology of science and engineering, and how they fit into the framework envisaged by the AECPD. We begin below by outlining the difference between engineering method and the use of craftsmanship principles based on intuition, but not proper science.

5.1 Engineering Intuition

(Sections 5 and 6 are heavily based on extracts from [29,30,31].) We typically think of *intuition* as the ability to know something without having to reason about it, or without being able to give a proper explanation, in the sense of science or engineering, of it. We have "intuitive" people in all walks of life - including engineering. In fact, we would go so far as to say that one role of a university engineering education is to try to foster "engineering intuition". However, we also claim that engineering intuition is not sufficient for the solution of engineering problems. We believe that engineering intuition guides the engineer in the choice of heuristics to try in the current problem. The engineering method, on the other hand, constrains the engineer to apply and test those heuristics in a very systematic way. So, intuition is not the difference between solving the problem and not solving it. Rather, it affects the speed with which the engineer arrives at a solution.

Exacerbating this intuition-science based engineering gap, it is our observation that there is a fundamental confusion between the scientifically and mathematically based practice of engineers and the day-to-day use of mathematics in the engineering *praxis*. This confusion results in discussions about "formal" versus "rigorous" (e.g., in the formal methods community), as if the dichotomy being explored was that between science/mathematics, on the one hand, and engineering, on the other. Actually, this difference resides completely in the science/mathematics camp. Only 'good" mathematicians and scientists are capable of doing rigorous mathematics. They know when they can leave out steps in proofs because they know or they are confident (they have good intuition) that the gaps can be filled. More typical mathematicians, scientists and engineers are not so good at doing this and have to rely more on not leaving such big gaps, or any at all. Thus less skilled mathematicians and scientists are capable of using only the formal, formulaic versions. Engineers use quite different scientific principles and mathematical techniques in their daily work [32]. It is with

respect to these practical uses of mathematics and science that engineers develop "intuitions". (The hydraulic engineer called in to resolve a knotty problem certainly recalled Bernoulli's equations when his intuition told him that the relation between the diameter of the tube and its effect on water flow is not linear, a recollection that enabled him to explain certain water shortages at Neuschwanstein castle in upper Bavaria, a shortage that the operatic stage designer who designed the castle could not explain.) In [29], Haeberer and Maibaum formulated a number of ad hoc principles that we would like to put forward and discuss. We will do this in the context of some ideas from epistemology that we believe can provide a framework for discussions of the nature of (Software) Engineering and for forming critical judgments of contributions to research and practice in the subject. The principles are:

1. Intuition is a necessary but not sufficient basis for engineering design.
2. Intuitions are very difficult to use in complex situations without well-founded abstractions or mental models. (The term "mental model" is used here as a synonym of a somewhat vague abstraction of a /emphframework in the Carnapian sense; see below.)
3. An engineer has our permission to act on intuition only when:
   - Intuitions can be turned into mathematics and science, and
   - Intuitions are used in the context of normal design processes (see below).
4. The abstractions, mental models, cognitive maps, ontologies used by engineers are not the same as those used by mathematicians and scientists.

5.2 Carnap's Statement View

Carnap's Statement View of Scientific Theories provides a setting for discussing these issues [33,34,35]. The primary motivation for the Statement View was to explain the language (and practice) of science. Haeberer and Maibaum, [29,30], adapted it to engineering and provided a framework to discuss issues such as intuition, method, and mathematics. According to Carnap, a scientific theory, relating some theory to observable phenomena, always has two disjoint subtheories: a theoretical one, not interpreted in terms of observable entities, and a purely observational one, related by a set of correspondence rules (often measurement procedures), which connect the two subtheories. According to Carnap's metaphilosophy, when we state some theory (or set of theories) to explain a set of observations stated in the observational language, therefore constructing an instance of the Statement View, we are putting in place a framework by making some ontological commitments. Once a framework is established, we automatically divide our (scientific) questions into two disjoint subsets, so-called internal questions (e.g., is it true that $E = mc^2$, and is it true that the halting problem is undecidable in the classical computability framework?) and so-called external questions (e.g., does Church's thesis provide a useful model of computation or not?). To assert that something is of utility, we must have in mind some task for which it is to be of utility.

Engineering, perhaps unlike science, is a normative subject. In our case, we are interested in discussing software engineering as a proper engineering discipline and use it as a basis for certifying its artefacts. That is, we want to ensure that the framework (in the Carnapian sense) is of utility in accomplishing the stated or intended purposes of engineering, generally, and software engineering, in particular. According to Vincenti [32], the day-to-day activities of engineers consist of *normal design*, as comprising the improvement of the accepted tradition or its application under 'new or more stringent conditions' ". He goes on to say: "The engineer engaged in such design knows at the outset how the device in question works, what are its customary features, and that, if properly designed along such lines, it has a good likelihood of accomplishing the desired task". Note the relationship to the definition of engineering above and Koen's view of engineering.

6 What Makes Software Engineering an Engineering Discipline?

The ongoing debate on engineering versus intuition motivated Haeberer and Maibaum to investigate the epistemology of software engineering, the role of mathematics in the software engineering curriculum, and the engineering nature of software engineering. This section is very heavily based on portions of that work, [29]. Mathematics is undoubtedly an essential tool in engineering. There are software engineers who still claim that mathematics is not necessary for producing software. Luckily, fewer and fewer are willing to say this. The real problem here is not the fact that mathematics is necessary, but that people tend to associate the mathematics required with that of theoretical computer science, rather than some appropriate *engineering mathematics*. In addition, many software engineers underestimate the importance of the role of heuristics (see Koen) and systematic method (see Vincenti), used in engineering to guide and constrain intuition.

Vincenti [32] argues the case for engineering being different, in epistemological terms and, consequently as *praxis*, from science or even applied science: "In this view, technology, though it *may apply* science, is not the same as or entirely *applied* science". GFC Rogers [36] argues that engineering is indeed different from science. He argues this view based on what he calls "the teleological distinction" concerning the *aims* of science and technology: "In its effort to explain phenomena, a scientific investigation can wander at will as unforeseen results suggest new paths to follow. Moreover, such investigations never end because they always throw up further questions. The essence of technological investigation is that they are directed towards serving the process of designing and manufacturing or constructing particular things whose purpose has been clearly defined. [...] It is also more limited, in that it may end when it has led to an adequate solution of a technical problem." He makes a further claim: "Because of its limited purpose, a technological explanation will certainly involve *a level of approximation that is certainly unacceptable in science* (our emphasis)." Going

back to the distinctions between the aims of science and engineering, we have, again from [36]: "We have seen that in one sense science progresses by virtue of discovering circumstances in which a hitherto acceptable hypothesis is falsified, and that scientists actively pursue this situation. Because of the catastrophic consequences of engineering failures - whether it be human catastrophe for the customer or economic catastrophe for the firm - engineers and technologists must try to avoid falsification of their theories. Their aim is to undertake sufficient research on a laboratory scale to extend the theories so that they cover the foreseeable changes in the variables called for by a new conception.

So science *is* different from engineering. Proceeding on this basis, we can ask ourselves what the *praxis* of engineering is (and ignore, at least for the moment, the specifics of scientific *praxis*). Vincenti defines engineering activities in terms of design, production and operation of artefacts. Of these, design and operation are highly pertinent to software engineering, while it is often argued that production plays a very small role, if any. In the context of discussing the focus of engineers' activities, he then talks about *normal design* as comprising "the improvement of the accepted tradition or its application under new or more stringent conditions' ". He goes on to say: "The engineer engaged in such design knows at the outset how the device in question works, what are its customary features, and that, if properly designed along such lines, it has good likelihood of accomplishing the desired task" (see [34].)

Another important aspect of engineering design is the organizing principle of hierarchical design: "Design, apart from being normal or radical, is also multilevel and hierarchical. Interesting levels of design exist, depending on the nature of the immediate design task, the identity of some component of the device, or the engineering discipline required." An implied, but not explicitly stated, view of engineering design is that engineers normally design *devices* as opposed to *systems*. A device, in this sense, is an entity whose design principles are well defined, well structured and subject to *normal* design principles. A system, the subject of *radical* design, in this sense, is an entity, which lacks some important characteristics making normal design possible. Examples of the former given by Vincenti are aeroplanes, electric generators, turret lathes; examples of the latter are airlines, electric-power systems and automobile factories. The software engineering equivalent of devices may include compilers, relational databases, PABXs, etc. Software engineering examples of systems may include air traffic control systems, mobile telephone networks, etc. It would appear that systems become devices when their design attains the status of being normal. That is, the level of creativity required in their design becomes one of systematic choice, based on well-defined analysis, in the context of standard definitions and criteria developed and agreed by engineers. This is what makes everyday engineering practice possible and reliable.

Let us now consider the particular characteristics of software engineering as a discipline. We want to address the question: "Is the knowledge used by software engineers different in character from that used by engineers from the conventional disciplines?" The latter are underpinned not just by mathematics, but

also by some physical science(s) - providing models of the world in terms of which artefacts must be understood. (The discussion above illustrates this symbiosis.) We might then ask ourselves about the nature of the mathematics and science underlying software engineering. It is not surprising, perhaps, that a large part of the mathematics underlying software engineering is formal logic.

Logic is the mathematics of concepts and abstractions. Software engineering may be distinguished from other engineering disciplines because the artefacts constructed by the latter are physical, whereas those constructed by the former are conceptual. There are some interesting and significant differences between the two kinds of mathematics and engineering mentioned above. One of these is that the real world acts as a (physical) constraint on the construction of (physical) artefacts in a way which is more or less absent in the science and engineering of concepts and abstractions. There seems to be a qualitative difference in the dimensions of the design space for software engineering as a result.

What distinguishes the theoretical computer science and software engineering dependence on logic is the day-to-day invention of theories (models) by engineers and the problems of size and structure introduced by the nature of the artefacts with which we are dealing in software engineering. Now, the relationship between the mathematics of theoretical computer science and that of (formal methods and) software engineering should be analogous to the difference between conventional mathematics and its application and use in engineering. As an example, program construction from a specification has a well-understood underlying mathematics developed over the last 25 years. (We are restricting our attention to sequential programs. Concurrency and parallelism are much less mature topics.) We might expect to find a CAD tool for program construction analogous to the "poles and canvas" model used in electronics for the design of filters. Instead, what we find is just a relaxation on the exhaustiveness requirement, i.e., we can leave out mathematical steps (proofs of lemmas) on the assumption that they can be filled in if necessary, the so-called rigorous approach. Where is the abstract model (analogous to the "poles and canvas" one) that encapsulates the mathematics and constrains manipulation in a (mathematically / scientifically) sensible manner?

6.1 An Epistemological Framework for Software Engineering

As Carnap (and others) have pointed out, an ontological framework, cannot be said to be correct or incorrect, it can only be of some utility, or not. Hence, in discussing a framework for software engineering, we are left with the task of convincing our colleagues that the proposed framework will be of some utility. We outline some details of the software engineering framework we proposed in terms of Figure 5, illustrating that we can give the diagram a semantics. That is, all the objects and relationships and processes denoted by the diagram could be given exact, mathematical/scientific definitions. (We say "could" because some of the relationships are presently the subject of research!) Nor do we claim that this is the only framework of utility for software engineering. (We only induce the reader to think about it as to be the last word in software engineering frameworks

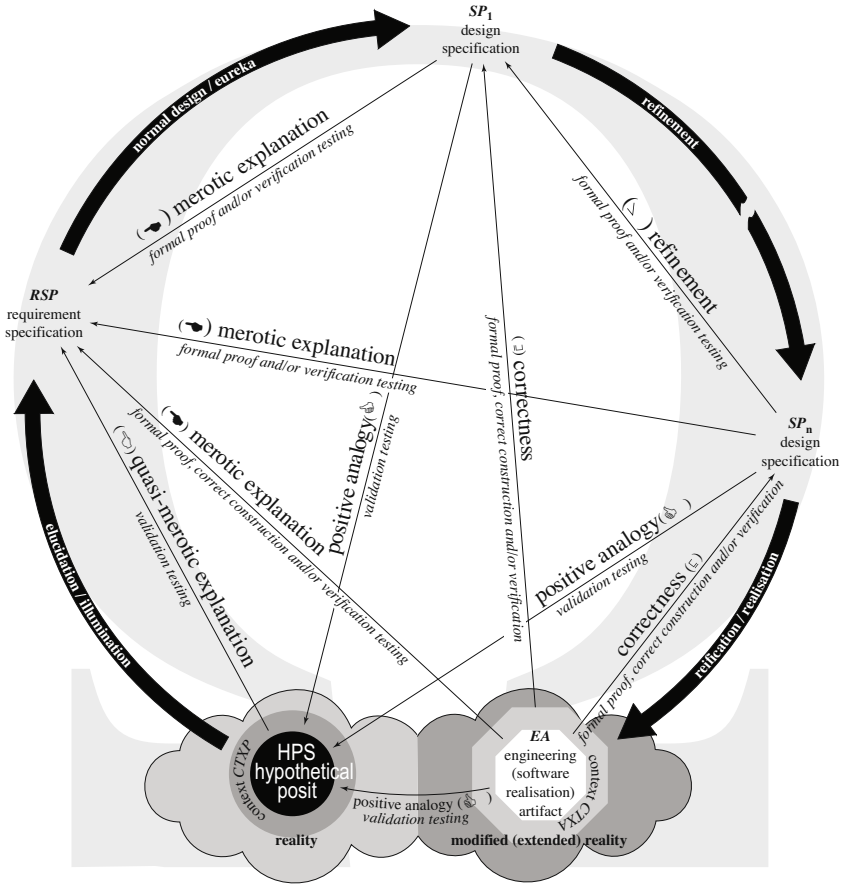


Fig. 5. Carnapian Framework for Software Engineering from [29]

by means of the background omega letter!) There may be others, more or less detailed, that are of equal utility. Actually, to assert that something is of utility, we must have in mind some task for which it is to be of utility. In our case, we are interested in making software engineering a proper engineering discipline (see, e.g., [30]) and supporting the practice of certification. Superficially, the elements of the diagram (objects and relationships) are just a more or less detailed version of diagrams used to represent the development process of software systems from conception to final realization as an executable system. As examples to illustrate that the elements of the diagram can be formalized, we give the following definitions: *Correctness* is a relation between two constructed artefacts asserting that the properties required by one are preserved in the other. Preservation of properties may be mediated by translation (between ontologies). Also, preservation does not exclude the inclusion of new (consistent) properties. *Validation* is the activity of determining, by means of experiments (i.e., testing), whether or

not we are constructing the appropriate positive analogy. *Positive analogy* is a relation between two entities (frameworks) consisting of a map between the two underlying ontologies (an interpretation between languages), which correlates positively (in the sense of essential and non-negative) properties of the source of the mapping with the positive properties of the target. We call the source an *iconic* model of the target. *Testing* is the application of tests. A *test* is an experiment to determine if some entity may have (can be assumed to have) some ground property (in the sense of logic).

We can use the framework to demonstrate the necessity of testing. We say that a relation is *epistemological* if it cannot, in principle, be formally (i.e., mathematically) corroborated. Hence, whether the relation holds or not is inherently contingent. That is, the existence of the relationship requires some form of testing (or experiment, in the sense of science) for its establishment. Despite its logical character, the truth of a logical relation is often checked by verification testing, in which case the character of this truth becomes contingent. The truth of an epistemological relation cannot be definitively established, just as a scientific theory cannot be "proved" once and for all.

6.2 Evidence and Measurement

Certification of any form requires evidence supporting the case for certification and judgements based on this evidence (the utility function mentioned above). The framework outlined above is intended to provide the foundation for building a framework for certification. The epistemological basis of science has established the principles and practice of using evidence in science. The adaptation to engineering ([29,30,31]), and software engineering in particular, enables us to apply informed judgements about proposals related to software development and certification. In particular, it provides a setting in which definitions of measurable attributes of software and their role in certification can be scientifically assessed. It also enables us to attempt assessments of their utility for the objectives of certification. It is on this basis that process-based approaches to certification should be rejected as insufficient to make certification judgements. The process-based assessment may well provide a statistical basis for confidence about the products of the process. But it does not provide sufficient levels of confidence about a particular product. The only way to obtain sufficient confidence about the product itself is to measure relevant product attributes and then make a judgement based on this evidence. Normally, in science, it is not sufficient for experiments to *usually* be successful in verifying some hypothesis about a theory. If an experiment fails to verify the hypothesis, there are only two possibilities: the experimental procedure was faulty, or the theory on which the hypothesis was based is false. (The former is probably the more usual cause for failure.) In the case of process-based predictions, there is a third possibility, namely that the process-based evidence was incorrect in relation to this particular product. Hence, the process-based approach does not pass the utility test: it fails to be reliable enough, and cannot, in principle, be "improved" to overcome this shortcoming. There is a lot more that could usefully be learned from the epistemology

of science and engineering. In particular, the concept of explanation in science ([37]) might form a useful basis for assessing the evidence produced by a manufacturer to support the licensing of a product. A basic question that needs to be asked during the assessment of the evidence is: Does the evidence provide a sufficiently good explanation (in this technical sense borrowed from epistemology) of the effectiveness and safety of the product to be accepted as a guarantee warranting certification? However, we shall not pursue this interesting topic here.

7 The Certification Initiative

In mid-2005, a number of researchers in academia and industry decided to start working on a *Certification Initiative*. The initiative was spearheaded by members of the Software Quality Research Laboratory (SQRL) at McMaster University in Canada, primarily Alan Wassyng, Tom Maibaum, Mark Lawford and Ryszard Janicki. Within a very short time, a small group of "Founding Members" was formed:

- SQRL faculty - McMaster University (Canada)
- Jo Atlee, University of Waterloo (Canada)
- Marsha Chechik, University of Toronto (Canada)
- Jonathan Ostroff, York University (Canada)
- Stefania Gnesi, ISTI-CNR (Italy)
- Connie Heitmeyer, NRL (USA)
- Brian Larson, Boston Scientific (USA)

The idea was to put software certification on the primary research agenda, and a number of activities have since resulted directly from this initiative.

7.1 The PACEMAKER Grand Challenge

With some encouragement from SQRL and Jim Woodcock, Brian Larson of Boston Scientific (Guidant), worked hard to release a natural language specification of a ten year-old model of a pacemaker. The specification forms the basis of a *Grand Challenge* to the software engineering community [38]. The PACEMAKER specification has also been used as a project for the first Student Contest in Software Engineering (SCORE) that is part of the 31^{st} International Conference on Software Engineering (ICSE 2009). A reference hardware platform was designed by students at University of Minnesota, supervised by Brian Larson, and Mark Lawford arranged to have 50 (slightly modified) PACEMAKER boards manufactured. They have been available through SQRL [39] for use in the PACEMAKER Grand Challenge, SCORE, and other academic endeavours.

The benefits we hope to realize from the PACEMAKER Grand Challenge and related activities are:

- Demonstrate the state-of-the-art in safety-critical software development.
- Provide a comparison of development methods.
- Develop product-focused certification methods.

7.2 The Software Certification Consortium

During 2007, SQRL researchers and Brian Larson spearheaded the formation of the Software Certification Consortium (SCC). Its purpose is to develop and promote an agenda for the certification of systems containing software (ScS), by forming a critical mass of industry, academic and regulatory expertise in this area. We held an inaugural meeting in August 2007, at SEI's Arlington location, and two further meetings in December 2007 (hosted by Mats Heimdahl, University of Minnesota) and late April 2008 (hosted by Austin Montgomery and Arie Gurfinkel, SEI). The current steering committee for SCC is:

- Richard Chapman (U.S. Federal Drug Administration)
- John Hatcliff (Kansas State University)
- Brian Larson (Boston Scientific)
- Insup Lee (University of Pennsylvania)
- Tom Maibaum (McMaster University)
- Bran Selic (Malina Software)
- Alan Wassyng (McMaster University)

A description of the goals of SCC, its objectives, and SCC's view of the major hurdles facing us in meeting those objectives was presented at SafeCert 2008 [40]. The goal of certification, SCC's goals and objectives are repeated below. The hurdles and their descriptions are also paraphrased below.

Goal of Certification - SCC: The *Goal of Certification* is to systematically determine, based on the principles of science, engineering and measurement theory, whether an artefact satisfies accepted, well defined and measurable criteria.

SCC Objectives:

(i) To promote the scientific understanding of certification for Systems containing Software (ScS) and the standards on which it is based;
(ii) To promote the cost-effective deployment of product-focused ScS certification standards;
(iii) To promote public, government and industrial understanding of the concept of ScS certification and the acceptance of the need for certification standards for software related products;
(iv) To investigate and integrate formal methods into ScS certification and development;
(v) To co-ordinate software certification initiatives and activities to further objectives i-iv above.

Goals to Achieve SCC Objectives: The *Primary Goals* are:

(i) Develop and document generic certification models that will serve as a framework for the definition of domain specific regulatory and certification requirements; and

(ii) Proof of concept: Develop and document software regulatory requirements that help both developers of the software and the regulators of the software in the development of safe, reliable software applications in specific domains.

A number of *Detailed Goals* were also identified: (i) Use existing software engineering and formal methods knowledge to develop appropriate evidence-based standards and audit points for critical software in specific domains, including hard real-time, safety-critical systems; (ii) Create software development methods that comply with the above standards and audit points for the development of critical software; and (iii) Research and develop improved methods and tools for the development of critical software.

Hurdles in Achieving Objectives: During the December 2007 SCC meeting, participants identified the following 9 hurdles. The first 4 of these were voted as the top 4 hurdles, in the order shown. The remaining 5 hurdles were not prioritized.

1. Clarity of regulator's expectation and method of communicating with the regulator. *Application developers do not know what to produce, and often have to pay consultants - who get it wrong.*
2. Lack of clear definition of evidence and how to evaluate it. *We know very little about the effectiveness of attributes and metrics related to dependability, and do not really understand how to combine different evidentiary artefacts.*
3. Poor documentation of requirements and environmental assumptions. *We need accurate and complete requirements in order to produce evidence of compliance. Poor requirements invariably lead to poor products.*
4. Incomplete understanding of the appropriate use of inspection, testing and analysis. *We do not know when to use inspection, testing and mathematical analysis to achieve specific levels of dependability.*

- No overarching theory of coverage that enables coverage to accumulate across multiple verification techniques. *In our opinion, this is the most important hurdle of all. It was not voted #1 simply because it was felt that we need to tackle easier hurdles first. We know of no single quality assurance technique that is solely sufficient for effective certification. Each of these techniques differs in strength of properties verified, types of behaviours covered, and the life-cycle stage in which they are most naturally applied. Sharing coverage across techniques via a single unified framework will enable the successes of one technique to reduce the obligations of associated techniques, and will clarify gaps in verification that must be filled by other techniques. The most convincing arguments of correctness will rely on being able to accurately state in quantitative ways how multiple verification techniques each contribute evidence of overall correctness.*
- Theories of coverage for properties like timing, tolerances as well as concurrent programs. *Structural coverage for testing plays a key role in development*

and certification of safety-critical software. Existing coverage measures fail to take into account properties such as timing and tolerance ranges for data values and the degree to which interleavings in concurrent computations are exercised As a result, even development efforts that succeed in achieving high levels of mandated coverage measures often fail to fully explore and validate common sources of program faults.

- Hard to estimate a priori the V&V and certification costs. *Currently, it is difficult to make a business case for the introduction of formal techniques, because it is difficult to estimate both the time required to carry out various forms of formal analysis and the reductions that can be obtained either in costs of the certification process itself or long-term costs associated with fewer defects found late in the development life-cycle, greater reuse in subsequent development of similar systems, fewer recalls of deployed systems, and decreased liability costs.*
- Lack of interoperable tools to manage, reason, and provide traceability.
- Laws, regulation, lawyers and politics. *Certification has legal implications, and as difficult as the technical problems may be, political considerations complicate the process immeasurably.*

8 Research Overview

Below, we list a number of broad research questions that we need to answer. In addition to these questions, the specific *hurdles* that were identified above in section 7.2 round out an initial research agenda for software certification.

- Is there a generic notion of certification, valid across many domains?
- What, if anything, needs to be adapted/instantiated in the generic model to make it suitable for use in a particular domain?
- What benefit do we achieve by creating product-specific software certification standards and processes?
- What simple process model is sufficient to enable the "faking" of real processes and providing a platform for evaluation by certification authorities?
- What is the difference between software quality, of a certain level, and certifiability?
- In what situations can we safely use process-based properties as a proxy for product qualities?
- If we have levels of certifiability, as in the Common Criteria, how does the mix of formal verification and testing change with the level?
- Since evaluating evidence about software is an onerous task, how can we assist evaluators to perform their tasks by providing tools? (Amongst examples of such tools may be proof checkers (to check proofs offered in evidence), test environments (to re-execute tests offered in evidence), data mining tools to find "interesting" patterns in artefacts, etc.)

9 Conclusions

Engineering methods are identifiable as those that are systematic, depend on theories and heuristics derived from relevant basic sciences, and rely on being able to measure relevant values in a repeatable way. Software engineering methods are moving - slowly - in that direction. Another important factor is the role of measurement in engineering and science. One of the major problems facing us is that we have not built or discovered adequate, meaningful metrics that can be used to measure attributes of software artefacts, either to support engineering methods, or more crucially, certification regimes. Unfortunately, existing software certification methods are primarily focused on evaluating the software development process that was used to develop the system being certified. This does not seem to qualify as an engineering approach to certification of software products. Almost all engineering certification regimes we have seen are product-focused. In any case, it seems that reliance on indirect evaluation of artefacts is a poor way of determining whether a product is effective, safe, and dependable.

We believe that software certification methods should be primarily product-focused. There are technical, social, commercial and political pressures being brought to bear on this movement. We also believe that there is growing agreement on this issue. We also think that there are good reasons why we should be examining whether or not we should be developing not just domain specific software certification methods, but even product specific, product-focused software certification methods. Interestingly, the FDA is also considering similar ideas [41], which would be a major change in direction for their certification regime.

The previous section briefly describes a research agenda that we believe will lead us to fundamental results that will aid in building new product-focused software certifications processes. In order to accomplish the goals of certification for software, we also have to undertake fundamental research on appropriate metrics for software and software design artefacts. We must develop significantly better engineering heuristics and methods, to make software development more reliable and repeatable, akin to classical engineering disciplines. Almost certainly, these heuristics and methods, and the accompanying certification regimes, will also be domain specific. This appears to be an inescapable attribute of engineering.

References

1. ASTM Standard F977: Standard Consumer Safety Specification for Infant Walkers. ASTM International, West Conshohocken, PA, USA (2000)
2. Regulatory Review and Recommendation Regarding Baby Walkers Pursuant to the Hazardous Products Act. Health Canada (April 2004)
3. General Principles of Software Validation; Final Guidance for Industry and FDA Staff. U.S. Dept. of Health and Human Services: FDA (January 2002)
4. Guidance for the Content of Premarket Submissions for Software Contained in Medical Devices; Guidance for Industry and FDA staff. U.S. Dept. of Health and Human Services: FDA (May 2005)
5. http://www.sei.cmu.edu/cmmi/ (March 2009)

6. Common Criteria for Information Technology Security Evaluation: Part 1: Introduction and general model, Version 3.1, Revision 1 (2006)
7. Common Criteria for Information Technology Security Evaluation: Evaluation methodology, Version 3.1, Revision 2 (2007)
8. Parnas, D.: The use of precise specifications in the development of software. In: IFIP Congress, pp. 861–867 (1977)
9. Heninger, K.L.: Specifying software requirements for complex systems: New techniques and their applications. IEEE Trans. on Soft. Engineering 6(1), 2–13 (1980)
10. Parnas, D.: Using Mathematical Models in the Inspection of Critical Software. In: Applications of Formal Methods, pp. 17–31. Prentice-Hall, Englewood Cliffs (1995)
11. Leveson, N.G., Heimdahl, M.P.E., Hildreth, H., Reese, J.D.: Requirements specification for process-control systems. IEEE Transactions on Software Engineering 20(9), 684–707 (1994)
12. Heimdahl, M.P.E., Leveson, N.G.: Completeness and consistency in hierarchical state-based requirements. IEEE Trans. on Soft. Eng. 22(6), 363–377 (1996)
13. Heitmeyer, C., Kirby Jr., J., Labaw, B., Archer, M., Bharadwaj, R.: Using abstraction and model checking to detect safety violations in requirements specifications. IEEE Transactions on Software Engineering 24(11), 927–948 (1998)
14. Crow, J., Di Vito, B.L.: Formalizing Space Shuttle software requirements: Four case studies. ACM Trans. on Soft. Eng. and Methodology 7(3), 296–332 (1998)
15. Archinoff, G.H., Hohendorf, R.J., Wassyng, A., Quigley, B., Borsch, M.R.: Verification of the shutdown system software at the Darlington nuclear generating station. In: International Conference on Control and Instrumentation in Nuclear Installations, Glasgow, UK, The Institution of Nuclear Engineers (May 1990)
16. Parnas, D.L., Asmis, G.J.K., Madey, J.: Assessment of safety-critical software in nuclear power plants. Nuclear Safety 32(2), 189–198 (1991)
17. Wassyng, A., Lawford, M.: Lessons learned from a successful implementation of formal methods in an industrial project. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 133–153. Springer, Heidelberg (2003)
18. Joannou, P., et al.: Standard for Software Engineering of Safety Critical Software. CANDU Computer Systems Engineering Centre of Excellence Standard CE-1001-STD Rev. 1 (January 1995)
19. Janicki, R., Wassyng, A.: Tabular representations in relational documents. Fundamenta Informaticae 68, 1–28 (2005)
20. Wassyng, A., Lawford, M.: Software tools for safety-critical software development. Software Tools for Technology Transfer (STTT) 8(4-5), 337–354 (2006)
21. McDougall, J., Viola, M., Moum, G.: Tabular representation of mathematical functions for the specification and verification of safety critical software. In: SAFECOMP 1994, pp. 21–30. Instrument Society of America, Anaheim (1994)
22. Wassyng, A., et al.: Choosing a methodology for developing system requirements. Ontario Hydro/AECL SD-2 Study Report (November 1990)
23. Easterbrook, S., Lutz, R., Covington, R., Kelly, J., Ampo, Y., Hamilton, D.: Experiences using lightweight formal methods for requirements modeling. IEEE Transactions on Software Engineering 24(1), 4–14 (1998)
24. Heitmeyer, C., Kirby, J., Labaw, B., Bharadwaj, R.: SCR*: A toolset for specifying and analyzing software requirements. In: Y. Vardi, M. (ed.) CAV 1998. LNCS, vol. 1427, pp. 526–531. Springer, Heidelberg (1998)
25. Shankar, N., Owre, S., Rushby, J.M.: PVS Tutorial. In: Computer Science Laboratory, SRI International, Menlo Park, CA (February 1993)
26. Rueß, H., Shankar, N., Srivas, M.K.: Modular verification of SRT division. Formal Methods in Systems Design 14(1), 45–73 (1999)

27. Parnas, D., Clements, P.: A rational design process: How and why to fake it. IEEE Trans. Software Engineering 12(2), 251–257 (1986)
28. Koen, B.: Definition of the Engineering Method. ASEE (1985)
29. Haeberer, A.M., Maibaum, T.S.E.: Scientific rigour, an answer to a pragmatic question: A linguistic framework for software engineering. In: ICSE 2001 Proceedings, pp. 463–472. IEEE Computer Society, Washington (2001)
30. Maibaum, T.: Mathematical foundations of software engineering: a roadmap. In: ICSE 2000 Proceedings, pp. 161–172. ACM, New York (2000)
31. Maibaum, T.: Knowing what requirements specifications specify. In: PRISE 2004, Conference on the PRInciples of Software Engineering, Technical Report, University of Buenos aires, keynote address in memory of Armando Haeberer (2004)
32. Vincenti, W.G.: What Engineers Know and How They Know It: Analytical Studies from Aeronautical History. The Johns Hopkins University Press, Baltimore (1993)
33. Carnap, R.: Empiricism, semantics, and ontology. Revue Internationale de Philosophie 11, 208–228 (1950)
34. Carnap, R.: The Methodological Character of Theoretical Concepts. In: Minnesota Studies in the Philosophy of Science, vol. II, pp. 33–76. U. of Minnesota Press (1956)
35. Carnap, R.: Introduction to the Philosophy of Science. Dover Publications, New York (1995)
36. Rogers, G.: The Nature of Engineering. The Macmillan Press Ltd., Basingstoke (1983)
37. Hempel, C.: Aspects of Scientific Explanation and Other Essays in the Philosophy of Science. The Free Press, New York (1965)
38. http://sqrl.mcmaster.ca/pacemaker.htm
39. http://www.cas.mcmaster.ca/wiki/index.php/Pacemaker
40. Hatcliff, J., Heimdahl, M., Lawford, M., Maibaum, T., Wassyng, A., Wurden, F.: A software certification consortium and its top 9 hurdles. In: Proceedings of SafeCert 2008. ENTCS (2008) (to appear)
41. Arney, D., Jetley, R., Jones, P., Lee, I., Sokolsky, O.: Formal methods based development of a PCA infusion pump reference model: Generic infusion pump (GIP) project, pp. 23–33 (June 2007)

Author Index