LNCS 5423

Michael Leuschel
Heike Wehrheim (Eds.)

Integrated Formal Methods

7th International Conference, IFM 2009
Düsseldorf, Germany, February 2009
Proceedings

Springer

Michael Leuschel   Heike Wehrheim (Eds.)

Integrated Formal Methods

7th International Conference, IFM 2009
Düsseldorf, Germany, February 16-19, 2009
Proceedings

Springer

Volume Editors

Michael Leuschel
Heinrich-Heine-Universität Düsseldorf, Institut für Informatik
Universitätsstraße 1, 40225 Düsseldorf, Germany
E-mail: leuschel@cs.uni-duesseldorf.de

Heike Wehrheim
Universität Paderborn
Fakultät für Elektrotechnik, Informatik und Mathematik
Warburger Straße 100, 33098 Paderborn, Germany
E-mail: wehrheim@uni-paderborn.de

Library of Congress Control Number: 2009920467

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

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-642-00254-4 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-00254-0 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12618477 06/3180 5 4 3 2 1 0

Preface

This volume contains the papers presented at the International Conference on integrated Formal Methods, iFM 2009, held on 16–19 February 2009 in Düsseldorf, Germany. The conference was the seventh in a series of conferences on integrated formal methods, with previous editions in York, Dagstuhl, Turku, Canterbury, Eindhoven and Oxford. The iFM conference series seeks to further research into the combination of different formal methods, both for modelling and analysis, covering all aspects from language design over verification techniques to tools and their integration into software engineering practice.

iFM 2009 received 55 submissions. Each submission was reviewed by at least three programme committee members. The submissions covered the whole spectrum of integrated formal methods, ranging from formal and semiformal modelling notations, semantics, verification, refinement and model transformations to type systems, logics, tools and case studies. The committee decided to accept 21 papers. The programme also included invited talks by David Basin, Michael Butler and Byron Cook. Collocated with the conference were two workshops (on "Integration of Model-based Methods and Tools" and "Formal Methods for SOA and Internet of the Future") and one tutorial (on "Contract Specification and Checking: Application to .NET and C") given by Shuvendru Lahiri and Francesco Logozzo (both from Microsoft Research).

We are grateful to all those involved in organizing the conference, producing the proceedings, reviewing the papers, and to the speakers and the attendees of iFM 2009. We also appreciate the support of EasyChair for managing the submission process.

December 2008

Michael Leuschel
Heike Wehrheim

Conference Organization

Programme Chairs

Michael Leuschel	University of Düsseldorf, Germany
Heike Wehrheim	University of Paderborn, Germany

Programme Committee

Eerke Boiten	University of Kent, UK
Einar Broch Johnsen	University of Oslo, Norway
Ana Cavalcanti	University of York, UK
Frédéric Dadeau	University of Besançon, France
Jim Davies	University of Oxford, UK
John Derrick	University of Sheffield, UK
Jin Song Dong	University of Singapore, Singapore
Neil Evans	AWE, UK
Martin Fränzle	University of Oldenburg, Germany
Andy Galloway	University of York, UK
Stefan Hallerstede	ETH Zürich, Switzerland
John Hatcliff	Kansas State University, USA
Marta Kwiatkowska	University of Oxford, UK
Frederic Lang	INRIA Rhône-Alpes, France
Michael Leuschel	University of Düsseldorf, Germany
Dominique Méry	LORIA Nancy, France
Stephan Merz	LORIA Nancy, France
Thomas Santen	Microsoft EMIC, Germany
Augusto Sampaio	University of Pernambuco, Brazil
Wolfram Schulte	Microsoft Research, USA
Graeme Smith	University of Queensland, Australia
Kenji Taguchi	NII, Japan
Helen Treharne	University of Surrey, UK
Ragnhild van der Straeten	University of Brussels, Belgium
Marina Waldén	Åbo Akademie University, Finland
Heike Wehrheim	University of Paderborn, Germany

Local Organization

Claudia Kiometzis
Michael Leuschel
Nadine Elbeshausen
Jens Bendisposto
Daniel Plagge

External Reviewers

Table of Contents

Invited Talks

Developing Topology Discovery in Event-B 1
Thai Son Hoang, Hironobu Kuruma, David Basin, and Jean-Raymond Abrial

Decomposition Structures for Event-B 20
Michael Butler

Taming the Unbounded for Hardware Synthesis 39
Byron Cook

Contributed Papers

Verifying UML/OCL Operation Contracts 40
Jordi Cabot, Robert Clarisó, and Daniel Riera

Property Specifications for Workflow Modelling 56
Peter Y.H. Wong and Jeremy Gibbons

Formal Verification Based on Guided Random Walks 72
Thang H. Bui and Albert Nymeyer

Parallel Processes with Real-Time and Data: The ATLANTIF Intermediate Format 88
Jan Stöcker, Frédéric Lang, and Hubert Garavel

Changing System Interfaces Consistently: A New Refinement Strategy for CSP||B 103
Steve Schneider and Helen Treharne

CSP with Hierarchical State 118
Robert Colvin and Ian J. Hayes

Predicate Abstraction in a Program Logic Calculus 136
Benjamin Weiß

Mechanised Translation of Control Law Diagrams into *Circus* 151
Frank Zeyda and Ana Cavalcanti

Realizability of Choreographies Using Process Algebra Encodings 167
Gwen Salaün and Tevfik Bultan

Modelling Divergence in Relational Concurrent Refinement 183
Eerke Boiten and John Derrick

SAL-Based Symbolic Scheduling in Time-Triggered Networks 200
Sebastian Voss, Maria Sorea, and Klaus Echtle

Incremental Reasoning for Multiple Inheritance 215
Johan Dovland, Einar Broch Johnsen, Olaf Owe, and Martin Steffen

Model Checking LTL Formulae in RAISE with FDR 231
Abigail Parisaca Vargas, Ana G. Garis, S. Lizeth Tapia Tarifa, and Chris George

An Introduction to Grammar Convergence 246
Ralf Lämmel and Vadim Zaytsev

Application of Graph Transformation in Verification of Dynamic Systems 261
Zarrin Langari and Richard Trefler

Formal Probabilistic Analysis of Stuck-at Faults in Reconfigurable Memory Arrays 277
Osman Hasan, Naeem Abbasi, and Sofiène Tahar

Challenges in the Specification of Full Contracts 292
Gordon J. Pace and Gerardo Schneider

Partial Order Reduction for State/Event LTL 307
Nikola Beneš, Lubos Brim, Ivana Černá, Jiri Sochor, Pavlina Vařeková, and Barbora Zimmerova

Dynamic Path Reduction for Software Model Checking 322
Zijiang Yang, Bashar Al-Rawi, Karem Sakallah, Xiaowan Huang, Scott Smolka, and Radu Grosu

Automatic Generation of Error Messages for the Symbolic Execution of EB3 Process Expressions 337
Jérémy Milhau, Benoît Fraikin, and Marc Frappier

Decompositional Petri Net Reductions 352
Astrid Rakow

Author Index 367

Developing Topology Discovery in Event-B*

Thai Son Hoang[1], Hironobu Kuruma[2], David Basin[1], and Jean-Raymond Abrial[1]

[1] Department of Computer Science, ETH Zurich
[2] Hitachi Systems Development Laboratory, Yokohama, Japan

Abstract. We present a formal development in Event-B of a distributed topology discovery algorithm. Distributed topology discovery is at the core of several routing algorithms and is the problem of each node in a network discovering and maintaining information on the network topology. One of the key challenges is specifying the problem itself. Our specification includes both safety properties, formalizing invariants that should hold in all system states, and liveness properties that characterize when the system reaches stable states. We establish these by appropriately combining proofs of invariant preservation, event refinement, event convergence, and deadlock freedom. The combination of these features is novel and should be useful for formalizing and developing other kinds of semi-reactive systems, which are systems that react to, but do not modify, their environment.

1 Introduction

We report here on a case study in critical system development using refinement. In our case study, we use the Event-B formalism [1] to specify and formally develop an algorithm for *topology discovery*, which is a problem arising in network routing. We proceed by constructing a series of models, where the initial models specify the system requirements and the final model describes the resulting system. We use the Rodin tool for Event-B [2] to prove that each successive model refines the previous one, whereby the resulting system is correct by construction.

The problem we examine is interesting for several reasons. First, it is a significant case study in specifying and developing distributed graph and routing algorithms. In routing protocols such as link-state routing [16], which is the basis for protocols such as OSPF [13,12] and OLSR [14], every router in the network must build a graph representing the network topology. In this graph, the vertices represent routing nodes and there is an edge from node a to node b if a can directly transmit data to b. Each node uses this graph to determine the shortest path to all other nodes, from which it constructs its routing table, which describes the best next hop to each destination. The main challenge in topology discovery is to ensure that the distributed construction of these graphs, as well as their updates after network changes, proceeds correctly. While there has been some work on using model checkers and theorem provers to verify properties of routing protocols (e.g., [6]), there have been relatively few case studies (e.g., [1,3,15]) in using formal methods to develop such protocols. Our work provides some insights on how this can be done.

* An extended version of this paper is [10] and a proof archive can be found at deploy-eprints.ecs.soton.ac.uk/31. Part of this research was carried out within the European Commission ICT project 214158 DEPLOY, www.deploy-project.eu/index.html. We thank Daniel Fischer, Matthias Schmalz, and Christoph Sprenger for their comments on drafts of this paper.

M. Leuschel and H. Wehrheim (Eds.): IFM 2009, LNCS 5423, pp. 1–19, 2009.

Second, as we will see, the problem of topology discovery is surprisingly nontrivial. The complexity is both in specifying the protocol's desired properties (what does it mean to "proceed correctly?") and in carrying out the development and proofs. This complexity comes from the fact that the protocol should function in dynamically changing environments. If we do not place constraints on the environment a priori (which we do not) then the actual topology may change faster than the nodes can propagate information about the changes they discover. For example, two nodes may be connected and not know it, but by the time they receive link information on this, they may be no longer connected. To address this, we present a novel approach to specifying and developing algorithms whose properties depend on the environment's dynamics. Our approach combines the use of *convergent events* in refinement (these are events that cannot take control of the system for ever) with a specification of deadlock freeness to specify the system's properties in stable system states.

Finally, our case study is representative of an important class of systems, which we call *(distributed) semi-reactive systems*. These are distributed systems where the environment is dynamically changing and, although the system cannot alter the environment, it must monitor and appropriately react to the changes in the environment. This includes, for example, distributed monitoring algorithms where the nodes must reach some kind of agreement about the environment's properties. Our approach suggests one way of developing systems in this general class.

2 Background on Event-B

Here we briefly describe the Event-B formalism; see [1,4] for further details. A development is a set of models described by contexts and machines. Contexts specify a model's static part, in terms of sets, constants, and axioms, whereas machines specify the dynamic part and correspond semantically to transition systems. A machine has variables, defining its state, and an initial state. The possible states are constrained by invariants. State transitions are described by events, which are guarded commands, each consisting of a guard and an action. The guard is a conjunction of predicates formalizing the necessary condition under which an event may occur, and the action describes how the state variables change when the event occurs. Semantically, an event denotes a relation (v, v') between the pre-state v (before the event) and the post-state v' after the event. We will later refer to the pairs $v \mapsto v'$ as instances of an event.

Machine refinement provides a means to introduce details about the dynamic properties of a model. Event-B's theory of refinement is closely related to that of Action Systems [5]. In particular, a concrete machine can refine another abstract machine, whereby their states are related by a (simulation) relation called a gluing invariant. Refinement is used to develop systems that are correct by construction. One specifies a series of machines $M_0, M_1, \ldots, M_n$, where each M_{i+1} refines M_i, the initial machines formalize the system's requirements, and the final machines formalize the system itself.

We have used the *Rodin Tool* [2] to create and analyze Event-B models. This tool generates proof obligations that ensure the correctness of the systems developed. These include: *invariant preservation* for establishing that invariants always hold; *refinement* between machines; and the *convergence* (termination) of sets of events (i.e., that the events in the set do not collectively diverge). Note that the convergence of some events

```
if DetectChange($x$,$v$) then
   UpdateLSDB($x$,$v$)
   UpdateSPFTree(LSDB)
   LSA $\leftarrow$ CreateLSA($x$,$v$)
   Broadcast(LSA)
end if
□
if Receive(LSA) then
   if IsFresh(LSA) then
      UpdateLSDB(LSA)
      UpdateSPFTree(LSDB)
      Broadcast(LSA)
   else
      Drop(LSA)
   end if
end if
□
Broadcast(LSDB)
```

Fig. 1. Link-state algorithm for node v (loop body)

cannot always be shown immediately and is then delayed to later refinements. In this case, the convergence of these events is *anticipated*.

3 Topology Discovery

In this section, we describe our requirements on the system and our assumptions on the environment for topology discovery. We begin by describing the problem and algorithm informally, in the context of link-state routing, which is one of its main applications.

3.1 Informal Description

Routing is the process of selecting paths through a network for sending data from a source to a destination. A path may require the data to travel over multiple hops, each hop being an intermediate router. At each router, data is forwarded using routing tables to select the next hop (the appropriate output port) based on the packet's destination address. It is the routing algorithm's task to build these routing tables. In link-state routing, this is done using several auxiliary data structures. In particular, each router maintains a link-state database (LSDB) that encodes its view of the topology of the communication network, i.e., the set of routers and the links between them. From its LSDB, a router computes a shortest path first (SPF) tree, using Dijkstra's algorithm [9]. The SPF tree is used to create the routing table: the next hop to some destination is simply the neighbor that constitutes the first link in the shortest path to that destination. Examples of routing algorithms that proceed this way include the Open Shortest Path First protocol (OSPF) [12,13] and (optimized) link-state routing [7,8].

In our case study, we focus on the important subproblem of *topology discovery*: discovering and maintaining local information about the network topology. This requires a distributed algorithm (protocol) since each node must construct its own local copy of the network topology. To do this, each node discovers changes in its own local communication environment and communicates them to other nodes. The nodes each individually build their own graphs, representing their local view of the global network topology.

To show how topology discovery is used in routing, Figure 1 presents a simplified view of link-state routing. The algorithm consists of an infinite loop, which runs on each node v. The loop's body nondeterministically chooses (represented by □) between three parts. From left-to-right these are: (1) detect and propagate changes; (2) receive and process changes; and (3) send information to neighboring nodes.

The first part describes how a node processes and propagates changes. Suppose a node v detects a change in the status of a link that joins some node x to v. The node v then adjusts its own link-state database (LSDB), which stores all topology graph nodes and edges. Afterwards, it updates its shortest path first (SPF) tree from the LSDB using Dijkstra's algorithm. Finally, it creates a link-state advertisement (LSA) describing the status (up or down) of the link from x to v, and starts flooding the network by broadcasting this to all of its neighbors. The second part describes a node's actions after receiving a link-state advertisement. If the LSA is fresh, then again the SPF tree is updated and the flooding is continued by sending the LSA to all neighbors. The third part states that a node v can, at anytime, start flooding the network by broadcasting information about its current link-state database. This can be implemented by v broadcasting an LSA describing the status of the link from x to y, for each pair of distinct nodes x and y. Alternatively, one message can be broadcast, describing the entire state of v's LSDB. In this case, the second part must be modified to also handle the reception of LSDBs.

These three parts implement basic link-state routing. If we are interested in pure topology discovery, it suffices to simply delete the two UpdateSPFTree statements. The resulting algorithm corresponds closely to what we will develop in Section 4.

A key point is the need for the third part, which initiates flooding even when no changes are present. This is required for two reasons. The first is to handle the possibility that LSAs are lost during communication, which can occur if a link goes down during message transit. The second reason is to handle the special case where disconnected parts of a network are reconnected. Suppose, for example, that the network is disconnected into two subnetworks S_1 and S_2, which each undergo changes and at some later time point become connected due to a link l coming up (i.e., l connects a node in S_1 with one in S_2). Just flooding both subnetworks with an LSA describing l being up is not enough for the nodes in S_1 to learn the topology of S_2 and vice versa. In actual link-state routing protocols, this third part, periodic flooding, occurs at fixed, relatively infrequent intervals. For example, in OSPF it takes place every 30 minutes.

Observe that the above algorithm description is still abstract and omits critical details. For example, nodes receive and propagate information at different times and hence a node may receive old LSAs containing invalid information about the network topology. How this is handled (e.g., using sequence numbers) and information is updated is not specified above. We must address precisely such details in our case study.

3.2 Requirements for Topology Discovery

As previously mentioned, it is surprisingly difficult to formulate the requirements for topology discovery. The protocol must operate in an *environment* where the status of links may change at any time. Moreover, the environment's behavior is out of the control of the protocol and not influenced by it (this is the notion of semi-reactive system, previously mentioned at the end of Section 1). If the environment changes sufficiently rapidly, then links reported as down may actually be up and vice versa. Hence the local LSDBs may bear little relationship to the actual network topology.

To tackle this problem, we focus on the limiting, and most important, case of the algorithm's behavior: its behavior when the environment is sufficiently quiescent. In this case, we expect that the local LSDBs will eventually converge (also called "stabilize" in the routing literature) to images of the actual global topology. Some care must be taken in precisely formalizing this, in particular to handle the previously mentioned problem that the network may not always be connected. In general, a node n can only learn about a link from a node a to its neighbor b when there is a path through the graph (representing the topology) from b to n.

Recall from basic graph theory that any graph can be decomposed into a collection of strongly-connected components. Our main system requirement is therefore:

System Requirement 1. If the environment is inactive for a sufficiently long time then for each strongly-connected component M, the local view (LSDB) of every node in M is in agreement with the actual topology, restricted to M.

Hence, when information about the system gained from link sensing (detecting communication neighbors) and communication stabilizes, each node has the correct view of the links between all nodes in its connected subnetwork.

We state one further requirement, which limits the possible local views of nodes during the protocol.

System Requirement 2. The local views of the nodes must be consistent with the past: a link listed as up is either up or was previously up.

This requirement rules out the case that a node concludes that a link is up that never was. So errors in the local topologies must effectively come from communication delays concerning status changes.

3.3 Environment Assumptions

Before developing a topology discovery algorithm, we must also be clear about our assumptions on the environment. We list them below.

Environment Assumption 1: There are only finitely many nodes.

Without this assumption, any notion of stability based on a hop-by-hop propagation of information would be unachievable.

Environment Assumption 2: There are directed, one-way links between some pairs of distinct nodes. Links may come up or go down at any time.

These links represent the ability to carry out directed (one-way) communication between two nodes. Links may be wired or wireless.

Environment Assumption 3: When there is a new link from node a to node b, then b is made aware of this. Likewise, when a link from a to b exists and is broken, b is also made aware of this.

We will refer to a link from a to b as either an *outward link* from a or an *inward link* to b. Assumption 3 reflects the ability to carry out "link sensing", whereby each node can sense its inward links. In practice, this must be realized by some kind of protocol, e.g., a must periodically announce its presence to b, or, in the bidirectional case, a handshake protocol initiated by b may be used. Note, as a result, that this assumption does not require that the receiver b immediately becomes aware of changes, but only eventually.

Environment Assumption 4: When a link goes down, any messages sent on it and not yet received are lost.

This reflects that communication is asynchronous. There is a delay (of unbounded length) between message transmission and reception, and messages can be lost during this time interval.

Environment Assumption 5: Nodes communicate by broadcasting whereby a may send a message to all b for which there exists a link from a to b.

Note that broadcasting is sufficient for topology discovery and is used during flooding. For other protocols, one might alternatively use point-to-point communication.

In the next section, we shall see how each of these requirements is formalized in the context of our Event-B development.

4 Formal Development

We now describe our development of topology discovery. We developed seven models. The initial models formalize our environmental assumptions and system requirements, whereas the subsequent models introduce design decisions for the resulting system.

Initial model: Specifies the protocol environment.
Refinement 1: Introduces the observer event for observing stable states and adds system events to model how nodes update their link information.
Refinement 2: Provides more details about link updates. Namely, a node updates information about its direct links or receives information about links from its neighbors.
Refinement 3: Introduces sequence numbers for tracking fresh link-state information.
Refinement 4: Uses message passing to transmit information about the status of links.
Refinement 5: Separates the events into two sets: the set of events that update link-state information and those events that discard it as being redundant. The idea is to prove the convergence of the events that update the link-state information.
Refinements 6: Introduces a variant for proving the convergence of some events.

Due to lack of space, we present below only selected parts of our formalization and omit proof details.

4.1 The Context and Initial Model

We begin by defining an Event-B context. In the context, we define the carrier set *NODES* of all network nodes and we axiomatize that it is finite. This formalizes **Environment Assumption 1**. Additionally, we define a (function) constant *closure* that, together with axioms, formalizes the transitive closure of binary relations between *NODES*. Note that ";" denotes forward relational composition.

axioms:
axm0_1 $\mathrm{finite}(\mathit{NODES})$
axm0_2 $\mathit{closure} \in (\mathit{NODES} \leftrightarrow \mathit{NODES}) \rightarrow (\mathit{NODES} \leftrightarrow \mathit{NODES})$
axm0_3 $\forall r \cdot r \subseteq \mathit{closure}(r)$
axm0_4 $\forall r \cdot \mathit{closure}(r); r \subseteq \mathit{closure}(r)$
axm0_5 $\forall r, s \cdot r \subseteq s \ \land \ s; r \subseteq s \ \Rightarrow \ \mathit{closure}(r) \subseteq s$

In our initial model, we formalize the behavior of the environment, where links (represented as pairs of nodes) may go up or down at any time. The variable $RLinks$ (R for real, i.e., actual links) represents the set of links that are currently up, whereas the variable $DLinks$ represents the set of links that have previously been up, but are now down. These sets are disjoint (**inv0_3**) since a link cannot be simultaneously both up and down. Note, however that we do not require that their union is the set of all links. This may be because two nodes are simply not communication neighbors or because their status has not yet been fixed. This set of "unknown" links is simply the complement of the set $RLinks \cup DLinks$.

```
variables:  $RLinks, DLinks$
```

```
invariants:
  inv0_1  $RLinks \in NODES \leftrightarrow NODES$
  inv0_2  $DLinks \in NODES \leftrightarrow NODES$
  inv0_3  $RLinks \cap DLinks = \varnothing$
```

Besides initializing $RLinks$ and $DLinks$ both to the empty set, there are two events: AddLink and RemoveLink. The former models that an arbitrary $link$ comes up. This link is then added to the set of $RLinks$ and removed from the set of $DLinks$ (if it is already there). The latter event handles the symmetric case. Note from the guards that if a link is in either set (i.e., its status is not unknown), then it has been up, at least once in the past.

```
AddLink
  any  $link$  where
    $link \notin RLinks$
  then
    $RLinks := RLinks \cup \{link\}$
    $DLinks := DLinks \setminus \{link\}$
  end
```

```
RemoveLink
  any  $link$  where
    $link \in RLinks$
  then
    $RLinks := RLinks \setminus \{link\}$
    $DLinks := DLinks \cup \{link\}$
  end
```

These events formalize **Environment Assumption 2**. Communication links are directed as the relations $RLinks$ and $DLinks$ are not necessarily symmetric.

4.2 The First Refinement

In our first refinement, we start to model the details of the protocol, although still very abstractly. In particular, we state that the link information stored at each nodes gets updated, although without yet specifying how.

We introduce two variables $rlinks$ and $dlinks$ with the following invariants. These two variables represent the current link-state information stored by each node.

```
invariants:
  inv1_1  $rlinks \in NODES \rightarrow (NODES \leftrightarrow NODES)$
  inv1_2  $dlinks \in NODES \rightarrow (NODES \leftrightarrow NODES)$
  inv1_3  $\forall n \cdot rlinks(n) \subseteq RLinks \cup DLinks$
  inv1_4  $\forall n \cdot dlinks(n) \subseteq RLinks \cup DLinks$
  inv1_5  $\forall n \cdot rlinks(n) \cap dlinks(n) = \varnothing$
```

The first two invariants formalize that each node stores its own local information (a binary relation between *NODES*) about the status of links. Moreover, if a node has some information about a link, then this link must be either currently up or down (i.e., not unknown). This is represented by the invariants **inv1_3** and **inv1_4**. The last invariant, **inv1_5**, states that a node cannot store contradictory information about the same link. Of course, different nodes can have different information about the same link.

Note that, together with the events AddLink and RemoveLink from the initial model, these invariants establish **System Requirement 2**. We have that a link can be in $DLinks$ iff it is removed with RemoveLink iff it was previously added to $RLinks$ with AddLink (since no other events change the state of $RLinks$ and $DLinks$) and therefore was previously up. Hence a link is only in $RLinks \cup DLinks$ if it is up (left disjunct) or was previously up (right disjunct).

One of the key aspects of our development strategy is to specify a so-called *observer event*. This event has no effect on this system state itself as its action is skip. Rather, its guard is used to define the notion of a *stable state* of the system.

```
stabilize
  status  $ordinary$
  when
    $\forall x, y \cdot x \mapsto y \in RLinks \Leftrightarrow x \mapsto y \in rlinks(y)$
    $\forall x, y \cdot x \mapsto y \in DLinks \Leftrightarrow x \mapsto y \in dlinks(y)$
    $\forall m, n \cdot m \mapsto n \in closure(RLinks) \Rightarrow$
      $(\forall k \cdot (k \mapsto m \in rlinks(n) \Leftrightarrow k \mapsto m \in rlinks(m)) \land$
           $(k \mapsto m \in dlinks(n) \Leftrightarrow k \mapsto m \in dlinks(m)))$
  then  skip  end
```

The first two guards require that every node y knows the correct status of all its inward links, i.e., y has detected all environment changes with respect to its inward links. The last guard requires that if there is a path from a node m to n, then n has the same (up/down) information as m for all inward links to m. Hence, the observer event fires in those states where nodes know the correct status of their neighbors and this status has already been propagated through the network along all outward links. Intuitively, in stable states, all nodes have the maximum knowledge of the environment that can be acquired from link sensing and communication. We say that the *system is in a stable state* when the observer event can fire.[1]

A central property that we proved is the following.

Theorem 1 (Stability implies correct local view). *If the system is stable, then for any strongly-connected component M in the network and any node n in M, n has the correct view of the status (up/down) of all links in M.*

We formulate this theorem in Event-B as follows, where $grdStabilize$ refers to the guard of the observer event.

[1] This notion of system stability is an instance of the general notion of a *stable system property* (see e.g., [11]), which is a property P of system states whereby if P is true of any reachable state s then P is true of all states reachable from s.

```
$grdStabilize$
  $\Rightarrow\ (\forall M \cdot (\forall f, l \cdot f \in M \land l \in M \land f \neq l \Rightarrow f \mapsto l \in closure(RLinks))$
    $\Rightarrow\ (\forall n \cdot n \in M$
      $\Rightarrow\ M \lhd rlinks(n) \rhd M = M \lhd RLinks \rhd M \quad \land$
         $M \lhd dlinks(n) \rhd M = M \lhd DLinks \rhd M\ ))$
```

Here, a set of nodes M defines a strongly-connected component of the graph whose edge relation is defined by $RLinks$, when for every distinct pair of nodes f and l in M, then $f \mapsto l \in closure(RLinks)$. The operators $\lhd$ and $\rhd$ respectively restrict the domain and the range of a relation to a set (here M, the strongly-connected component).

We proved this theorem using the Rodin tool. The theorem itself constitutes part of the proof of **System Requirement 1**. Namely, in a stable state, each node has the correct view of all links in its strongly-connected component. It still remains to be proved that this stable state will be reached whenever the environment is inactive for a sufficient long time period. We prove this in Section 4.8.

In this model, we also introduce two new events, addlink and removelink, which modify the link-state information of some node.

```
addlink
  status  $anticipated$
  any  $n, link$  where
    $n \in NODES$
    $link \in RLinks \cup DLinks$
  then
    $rlinks(n) := rlinks(n) \cup \{link\}$
    $dlinks(n) := dlinks(n) \setminus \{link\}$
  end
```

```
removelink
  status  $anticipated$
  any  $n, link$  where
    $n \in NODES$
    $link \in RLinks \cup DLinks$
  then
    $rlinks(n) := rlinks(n) \setminus \{link\}$
    $dlinks(n) := dlinks(n) \cup \{link\}$
  end
```

The event addlink abstractly models a node receiving information on a link directly from the topology. Specifically, the event nondeterministically selects a node n and a link $link$ with a known status. It then updates n's local information about $link$, ensuring that it is added to the set of real (up) links and removed from the set of down links. Perhaps counterintuitively, the event may add a link to $rlinks(n)$ that is actually down, i.e., that belongs to $DLinks$. This reflects a key aspect of our distributed algorithm: the information nodes receive about the environment may be out-dated. As noted previously, being in $RLinks \cup DLinks$ simply means the node has been up in the past. But by the time n receives information that $link$ is up, the link may actually be down. The second event removelink is analogous. At this level of refinement, addlink and removelink are *anticipated*. That is, we delay the proof that these events converge to subsequent refinements.

From now on, we concentrate on the refinement of addlink. The refinement of removelink can be found in our on-line development archive.

4.3 The Second Refinement

In this refinement, we specify more concretely how link information is updated in each node. There are two cases. The first case models a direct update by the hello event. The second case models an indirect update by the transfer_rlink event.

```
hello
  refines  $addlink$
  status  $convergent$
  any  $n, m$  where
    $m \mapsto n \in RLinks$
    $m \mapsto n \notin rlinks(n)$
  then
    $rlinks(n) := rlinks(n) \cup \{m \mapsto n\}$
    $dlinks(n) := dlinks(n) \setminus \{m \mapsto n\}$
  end
```

```
transfer_rlink
  refines  $addlink$
  status  $anticipated$
  any  $n, m, x, y$  where
    $x \mapsto y \in rlinks(m) \cup dlinks(m)$
    $n \neq y$
  then
    $rlinks(n) := rlinks(n) \cup \{x \mapsto y\}$
    $dlinks(n) := dlinks(n) \setminus \{x \mapsto y\}$
  end
```

The event hello models a node n discovering information (e.g., by receiving a "hello" message) from a node m with an outward link to n. This event refines the abstract event addlink, where the abstract parameter $link$ is represented by the mapping $m \mapsto n$. The event transfer_rlink models a node n receiving information about a link $x \mapsto y$ from some node m, which is not necessarily a neighbor. The guard $n \neq y$ indicates that this is an indirect update, that is, $x \mapsto y$ is not an inward link of n. This refines the abstract event addlink, where the abstract parameter $link$ is represented by the mapping $x \mapsto y$.

The link-state information for down links is modeled analogously by the events goodbye and transfer_dlink, which are omitted here. Together, hello and goodbye formalize **Environment Assumption 3**.

At this stage, we also prove the convergence of the hello and goodbye events and we will prove the convergence of the transfer_rlink and transfer_dlink events in the next refinement, that is, they are anticipated at this point. By decomposing the convergence proof into different refinements we can simplify the proof by decomposing the events into two different subsets and then considering these subsets individually. Note that when proving the convergence, we still have the obligation of proving that the anticipated events do not increase the new variant. Taken together, these steps imply that the events reduce a composite variant, built from the lexicographic combination of the variants used in the two proofs.

We prove convergence by showing that these two events always decrease a *variant*, which is a set-valued expression. In this case the variant is

$$\{m \mapsto n \mid m \mapsto n \in RLinks \setminus rlinks(n)\} \cup$$
$$\{m \mapsto n \mid m \mapsto n \in DLinks \setminus dlinks(n)\} .$$

This is the set of inward links to n, where n has incorrect information. Informally, since the hello and goodbye events both provide correct information about one inward link of a node, they decrease the variant. Since the set of *NODES* is finite, this variant is also finite and thus well-founded.

4.4 The Third Refinement

In the next two refinement steps, we model communication between nodes. This is in contrast to the last step where nodes update their link information directly using the link information of other nodes, which is of course not realizable in a distributed system.

Before modeling communication, we first model how nodes track which information is fresh, i.e., whether the link information received is new or old. Namely, we introduce a new variable, $seqNum \in \mathit{NODES} \rightarrow (\mathit{NODES} \times \mathit{NODES} \rightarrow \mathbb{N})$ representing the sequence number stored at each node for each link. We omit listing here the invariants for $seqNum$. Moreover, to reason about the convergence of transfer_rlink and transfer_dlink, we introduce an auxiliary variable msg that "measures" the convergence of the event. This variable will not be used in the guards of the event, that is, it will not affect the execution of the events, hence we can safely remove this variable in the subsequent refinement.

In the initialization event, the sequence number for all links is set to 0 and msg is empty. The sequence number for a particular node and link first takes on a positive value after a direct update (e.g. in the hello event).

```
hello
  **refines**  $hello$
  **any**  $n, m$  **where**
    $m \mapsto n \in RLinks$
    $m \mapsto n \notin rlinks(n)$
  **then**
    $rlinks(n) := rlinks(n) \cup \{m \mapsto n\}$
    $dlinks(n) := dlinks(n) \setminus \{m \mapsto n\}$
    $seqNum(n) := seqNum(n) \mathbin{<\!\!+} \{(m \mapsto n) \mapsto seqNum(n)(m \mapsto n) + 1\}$
    $msg := msg \cup (\{m \mapsto n \mapsto seqNum(n)(m \mapsto n) + 1\} \times (\mathit{NODES} \setminus \{n\}))$
  **end**
```

The only difference with the abstract version is the last two assignments, which increment the sequence number ($\mathbin{<\!\!+}$ denotes relation overriding) and update msg. Since the event's guard is unchanged and the additional assignment modifies only a new variable, this clearly refines the corresponding abstract hello event. Once new information is detected by n, this information must be propagated to all the other nodes in the network.

For indirect updates, the sequence number for a particular link is not updated, but simply passed from one node to another.

```
transfer_rlink
  **refines**  $transfer\_rlink$
  **status**  $convergent$
  **any**  $n, m, x, y, sn$  **where**
    $m \mapsto n \in RLinks$
    $sn \leq seqNum(m)(x \mapsto y)$
    $seqNum(n)(x \mapsto y) < sn$
    $\forall k \cdot seqNum(k)(x \mapsto y) = sn \Rightarrow x \mapsto y \in rlinks(k)$
  **then**
    $rlinks(n) := rlinks(n) \cup \{x \mapsto y\}$
    $dlinks(n) := dlinks(n) \setminus \{x \mapsto y\}$
    $seqNum(n) := seqNum(n) \mathbin{<\!\!+} \{(x \mapsto y) \mapsto sn\}$
    $msg := msg \setminus \{x \mapsto y \mapsto sn \mapsto n\}$
  **end**
```

Compared to the abstract version of the event, there is an additional parameter, sn, for the sequence number associated with the link-state information. This sequence number sn is no more than the sequence number that m has for the same link. The reason is that the original message came from m and sequence numbers are never decreased.[2] The sequence number sn is (strictly) greater than n's sequence number for the same link, that is, n only updates its local state with new information. The last guard states that for any node k with the same sequence number for the same link $x \mapsto y$, that link is in the set of up links for k. This ensures that there will be no conflicting information in the network. Note that this guard cheats in the sense that it cannot be directly implemented. This cheating will be eliminated in a subsequent refinement. The additional assignments in the event's action, with respect to the abstract version, update n's sequence number for the link $x \mapsto y$ and remove this information from the set msg.

We also proved the convergence of the transfer_rlink and transfer_dlink events. The variant is just msg. This, together with the convergence proof from the last refinement, shows that the events hello, goodbye, transfer_rlink, transfer_dlink decrease a combined lexicographic variant.

The guard of the observer event stabilize (from the first refinement) is also refined using information about sequence numbers. It becomes:

```
stabilize
  when
    $\forall x, y \cdot x \mapsto y \in RLinks \Leftrightarrow x \mapsto y \in rlinks(y)$
    $\forall x, y \cdot x \mapsto y \in DLinks \Leftrightarrow x \mapsto y \in dlinks(y)$
    $\forall n1, n2, link \cdot n1 \mapsto n2 \in RLinks \Rightarrow$
      $seqNum(n1)(link) \leq seqNum(n2)(link)$
  then  skip  end
```

The first two guards are unchanged. What is new is the last guard, which states that for any pair of nodes $n1$ and $n2$, and link $link$, if $n1$ has a direct communication link to $n2$, then $n2$'s information about $link$ is not older than $n1$'s.

4.5 The Fourth Refinement

We now model communication. We first remove the auxiliary variable msg. We also remove the assignments that modify msg from the events hello and goodbye. We then introduce three variables: $SChan$ of type $(NODES \times NODES) \rightarrow ((NODES \times NODES) \rightarrow \mathbb{N})$ and $RChan$ and $DChan$, both of type $(NODES \times NODES) \rightarrow (NODES \leftrightarrow NODES)$. For each pair of nodes, the link-state information is a relation between *NODES*, formalizing the set of pairs of nodes on the communication channel. For all nodes m and n, $RChan(m \mapsto n)$ (respectively, $DChan(m \mapsto n)$) is the set of up (down) link information that is transferred from m to n. The channel $SChan$ associates sequence numbers to the links in the link-state channels. Thus $SChan(m \mapsto n)$ stores information about the sequence numbers that are in transit from m to n.

[2] However, sn can differ from m's sequence number, since during the time for the message to reach n, m can in the meantime update its sequence number for the same link.

Communication between nodes uses the above channels, so the abstract events for transferring link information (namely, transfer_rlink and transfer_dlink) must each be split into a pair of events for sending and receiving information. The following diagram illustrates what happens. First, the node m *sends* the information to the channels and afterwards the node n *receives* information from the channels. In our development, each transfer event is refined by a receive event and we add a new send event, which therefore refines skip. In our diagram, the top part is the abstraction (skip and *transfer*) and the bottom part is the refinement (i.e., *send* and *receive*).

$$\begin{tikzcd}
m \arrow[r, "\mathsf{skip}"] & {} \arrow[r, "transfer"] & n \\
m \arrow[r, "send"] & \boxed{\text{channels}} \arrow[r, "receive"] & n
\end{tikzcd}$$

Below is the description of the new event for sending information about an up link from m to n.

```
send_rlink
  status  $anticipated$
  any  $m, n, link$  where
    $m \mapsto n \in RLinks$
    $SChan(m \mapsto n)(link) = 0$
    $link \in rlinks(m)$
  then
    $SChan(m \mapsto n) := SChan(m \mapsto n) \Leftarrow \{link \mapsto seqNum(m)(link)\}$
    $RChan(m \mapsto n) := RChan(m \mapsto n) \cup \{link\}$
  end
```

For a node to send information about certain $link$, this event requires that the information about the same $link$ from the last send has been received. This is formalized by the guard stating that the corresponding sequence number in the channel is 0. The information is then sent by placing it on the outward links from m to n. The guard $m \mapsto n \in RLinks$ (i.e. the link from m to n is currently up) formalizes **Environment Assumption 5**.

The abstract transfer_rlink is refined to specify the following event receive_rlink.

```
receive_rlink
  refines  $transfer\_rlink$
  any  $m, n, x, y$  where
    $seqNum(n)(x \mapsto y) < SChan(m \mapsto n)(x \mapsto y)$
    $x \mapsto y \in RChan(m \mapsto n)$
  then
    $rlinks(n) := rlinks(n) \cup \{x \mapsto y\}$
    $dlinks(n) := dlinks(n) \setminus \{x \mapsto y\}$
    $seqNum(n) := seqNum(n) \Leftarrow \{(m \mapsto n) \mapsto SChan(m \mapsto n)(x \mapsto y)\}$
    $SChan(m \mapsto n) := SChan(m \mapsto n) \Leftarrow \{(x \mapsto y) \mapsto 0\}$
    $RChan(m \mapsto n) := RChan(m \mapsto n) \setminus \{x \mapsto y\}$
  end
```

The link-state information is retrieved from the channels from m to n. Here, the abstract parameter sn is refined as $SChan(m \mapsto n)(x \mapsto y)$. The refinement of transfer_dlink to receive_dlink is analogous.

Note that the event receive_rlink receives only genuinely new messages. Hence it is necessary to introduce a *complement* event that discards obsolete information, both for up and down links. Another reason for introducing discard events is that, without them, we would not be able to prove the deadlock freeness property in the next refinement level. Below is the event for discarding information about an up link (the new event discard_dlink is analogous).

```
discard_rlink
  status  $anticipated$
  any  $m, n, link$  where
    $SChan(m \mapsto n)(link) \leq seqNum(n)(link)$
    $link \in RChan(m \mapsto n)$
  then
    $SChan(m \mapsto n) := SChan(m \mapsto n) \Leftarrow \{link \mapsto 0\}$
    $RChan(m \mapsto n) := RChan(m \mapsto n) \setminus \{link\}$
  end
```

The link-state information is obsolete since the node has already received more recent information about $link$ in the channel. Hence, the information is simply discarded from the channel. This new event refines skip since the actions only effect the new variables, $SChan$ and $RChan$.

Now that we have explicitly introduced communication, we refine the environment event RemoveLink to account for **Environment Assumption 4**. That is, when a link goes down, any messages sent on it and not yet received are lost.

```
RemoveLink
  refines  $RemoveLink$
  any  $link$  where
    $link \in RLinks$
  then
    $RLinks := RLinks \setminus \{link\}$
    $DLinks := DLinks \cup \{link\}$
    $SChan := SChan \Leftarrow (\{link\} \times \{NODES \times NODES \times \{0\}\})$
    $RChan(link) := \varnothing$
    $DChan(link) := \varnothing$
  end
```

This trivially refines the abstract RemoveLink event since the guard is unchanged and the new assignments only modify new variables.

Note that at this point all the events can be straightforwardly implemented in a distributed system. That is, the events no longer "cheat" and perform tests or actions that would not be algorithmically realizable.

4.6 The Fifth Refinement

Our machine in the fourth refinement constitutes a (high-level) protocol implementation. However, we have not yet established the convergence of the events send_rlink and discard_rlink (and correspondingly for $dlink$). There is a good reason for this: these events do not converge and should not converge. As we saw in Figure 1 (third part), each node periodically broadcasts information about its LSDB and its neighbors repeatedly receive this information, even when it is not new. What we prove then is that the system eventually does reach a stable state (assuming that the environment does not change), despite continually broadcasting and receiving redundant information.

To prove this, we shall partition these four non-convergent events each into two parts: a convergent and divergent part. We accomplish this by defining a restricted local notion of stability, called neighbor stability, and showing that the neighbor-stable parts diverge and, conversely, the neighbor-unstable parts converge.

Given a link $link$ and a link from m to n, we say the information about $link$ is *neighbor stable* from m to n if n's sequence number for $link$ is at least as large as m's. This means that the information about $link$ in m does not need to propagate to n and therefore further information coming from m about $link$ will not change this neighbor-stable status. Using this notion, we can restate the third guard of the observe event stabilize (from Section 4.4) as follows: Any $link$ is neighbor-stable for any up link from m to n.

We now partition the events by adding either the guard $seqNum(m)(link) \leq seqNum(n)(link)$ or its complement. For example, we partition send_rlink into the two events send_rlink_stable and send_rlink_unstable. For send_rlink_stable we add the above guard and for send_rlink_unstable we add the complement as a guard. We partition the other three events discard_rlink, send_dlink, and discard_dlink similarly. Note that we must partition the discard events as information must also be discarded in neighbor-unstable states. The reason for this is that communication is asynchronous and therefore information may be sent in a stable state but received in an unstable state.

Given this partition, we prove the convergence of the events send_rlink_unstable and send_dlink_unstable using the variant

$$\{m \mapsto n \mapsto link \mid SChan(m \mapsto n)(link) \leq seqNum(n)(link)\}\,.$$

This denotes the set of old messages on all channels. We will prove the convergence of discard_rlink_unstable and discard_dlink_unstable in the next refinement level and hence they act as anticipated events here.

In this refinement step, we also proved the following theorem about the deadlock freeness of a set of events. Namely, the guard of the event stabilize is equivalent to the negation of the disjunction of the guards of the following eight events: hello, goodbye, send_rlink_unstable, send_dlink_unstable, receive_rlink, discard_rlink_unstable, receive_dlink, and discard_dlink_unstable. Hence, if none of these eight events is enabled, then stabilize is enabled and the system is therefore in a stable state.

Moreover, we also proved theorems stating that the four events send_rlink_stable, send_dlink_stable, discard_rlink_stable, and discard_dlink_stable maintain the system's stable state, that is, if we assume that the state before the event execution is stable, we have to prove that the state after the event execution is also stable. However, $stable$

refers to $RLinks$, $DLinks$,$rlinks$, $dlinks$, and $seqNum$ only, whereas our events (send_rlink_stable, send_dlink_stable, discard_rlink_stable, and discard_dlink_stable) only modify the information in the channels, i.e., $SChan$, $RChan$, and $DChan$, so the above events will maintain the stable state.

4.7 Sixth Refinement

In this refinement step, we prove the convergence of the discard_rlink_unstable and discard_dlink_unstable using the variant

$$\{m \mapsto n \mapsto link \mid SChan(m \mapsto n)(link) \neq 0\} \cap$$
$$\{m \mapsto n \mapsto link \mid seqNum(n)(link) < seqNum(m)(link)\}\,.$$

4.8 Partial Convergence Implies Stability

In contrast to the development of terminating programs, we now only prove the convergence of a subset of the events. Nevertheless, we show that this is adequate to establish **System Requirement 1**. Namely, if the environment is inactive for a sufficiently long time, then for each strongly-connected component M, the local view of every node in M is in agreement with the actual topology, restricted to M.

First, we introduce the notion of a *run* of Event-B together with a strong-fairness assumption. A run of an Event-B model is an infinite sequence of states obtained from an initial state by executing events of the model. We call a run *strongly fair* with respect to a set of events E if it respects the following *strong-fairness* assumption with respect to E: if an event from E is enabled infinitely often, then it will be taken infinitely often. This assumption will hold for any reasonable implementation of topology discovery.

At the last refinement, the set of events can be divided into the following groups.

1. A set of environment events $Env = \{Env_1, \ldots, Env_l\}$. In our case, there are just the two events AddLink and RemoveLink.
2. An observer event Obs. This observer event has skip as its action and its guard specified that the system is in stable state. Hence it is of the form:

   when *stable* **then** skip **end**

   In our development, this is the stabilize event.
3. A set of convergent events $CE = \{CE_1, \ldots, CE_m\}$. In our development, the convergent events are hello, goodbye, send_rlink_unstable, send_dlink_unstable, receive_rlink, discard_rlink_unstable, receive_dlink, and discard_dlink_unstable.
4. A set of divergent events $DE = \{DE_1, \ldots, DE_n\}$. These events are send_rlink_stable, send_dlink_stable, discard_rlink_stable, and discard_dlink_stable.

We will now prove the following theorem:

Theorem 2 (System Stabilizes). *Assume that the following propositions hold:*

i) Deadlock-freedom for the observer event Obs and convergent events CE. In particular,

$$stable \Leftrightarrow \neg(Grd(CE_1) \vee \cdots \vee Grd(CE_m))\,.$$

ii) The events in CE converge using a well-founded variant V.
iii) The events in DE do not increase V.
iv) The events in DE preserve $stable$. By this we mean that none of the DE events disable the guard of Obs.
v) The events in CE are strongly fair.

Then if the environment is eventually quiescent (i.e., at some point no environment events Env_1, ..., Env_l from the first group occur) then the system will eventually reach a stable state and remain in this state.

In our case, we are assuming Proposition (v), and the other propositions have already been previously proved.[3] Our proof of Theorem 2 is by contradiction and proceeds as follows. Assume that there is a strongly fair run R with a quiescent suffix, but which never reaches a stable state. Then there must be infinitely many i such that $R(i)$ does not satisfy "stable". Let r be a quiescent suffix of R. By Proposition (i), there are infinitely many states such that some event in CE is enabled. By the fairness assumption, Proposition (v), the events in CE must be taken infinitely often on r. Since there are no environment events and by Proposition (ii) all events in CE decrease the variant, whereas by Proposition (iii), other system events (i.e., Obs and DE) do not increase the variant V, the variant V decrease infinitely often in r. This contradicts the well-foundedness of V. Therefore, all strongly fair runs with a quiescent suffix eventually reach a stable state. Moreover, once in a stable state, all the events in CE are disabled and, by Proposition (iv), the events in DE preserve the stable state. Together with the fact that event Obs does not change the state (its action is skip), it follows that the system stays in the stable state. This concludes our proof. Note that this proof is a traditional "paper and pencil proof", rather than a proof using the Rodin tool.

The system referred to in the theorem statement is the machine M_5 given by the 5th refinement, rather than the machine M_4 from the 4th refinement, which is our implementation. However, M_5 simply partitions four of M_4's events. Therefore the proof of Theorem 2 just given for M_5 can be naturally mapped to M_4. Namely, the partition of M_4's events into stable and unstable events in M_5 gives rise to a partition of their instances. Therefore Theorem 2 also holds for M_4 if we restate the fairness assumption in Theorem 2 as follows: "If an instance of event is enabled infinitely often, then it will be taken infinitely often."

Finally, recall Theorem 1, proved in Section 4.2, which states that in a stable state, each node has the correct view of all links in its strongly-connected component. It follows from this and Theorem 2 that the system M_4 satisfies **System Requirement 1**.

4.9 Summary — Proof Statistics

In Table 1 we give proof statistics of the development in the Rodin Tool. These statistics measure the size of the model, the proof obligations generated and discharged by the Rodin Platform, and those interactively proved. Note that there are many proof obligations in the 4th refinement due to the introduction of three different channels. In order to

[3] We proved Propositions (i) and (iv) in the 5th refinement and proved Propositions (ii) and (iii) in the 2nd, 3rd, 5th, and 6th refinements.

guarantee the correctness using these channels, various invariants must be established. Moreover, our formal model of these channels uses high-order functions. Given the current state of the Rodin platform, this results in a high number of interactive (manual) proofs. Also, most of the proofs in the 5th and the 6th refinements are interactively discharged. The main reason for this is the lack of appropriate automatic support in the tool for reasoning about set comprehension, disjunctions, and strict subsets.

Table 1. Proof statistics

Model	Number of Proof Obligations	Automatically Discharged	Interactively Discharged
Context	3	0(0%)	3(100%)
Initial Model	9	9(100%)	0(0%)
1st Refinement	31	26(84%)	5(16%)
2nd Refinement	30	23(77%)	7(23%)
3rd Refinement	74	37(50%)	37(50%
4th Refinement	159	79(50%)	80(50%)
5th Refinement	44	7(16%)	37(84%)
6th Refinement	8	0(0%)	8(100%)
Total	358	181(51%)	177(49%)

5 Conclusions

We have presented a case study in formally developing a distributed topology discovery algorithm in Event-B. Our approach to formalizing and reasoning about stable states should be applicable to other semi-reactive systems, including other routing algorithms. Our approach is novel in how it combines refinement with arguments about convergence and disjointness of events to specify liveness properties about the system eventually stabilizing and properties of the resulting stable state.

We have presented a single development of topology discovery. In actuality, we formalized several different developments, each highlighting a different aspect of the problem, making different assumptions about the environment, and establishing different properties. For example, we first considered the case where the environment is static and we developed a terminating algorithm satisfying a strong post-condition. We also considered the case where the environment is dynamic and not necessarily stabilizing. There we had the idea of augmenting the environment with some history ($DLinks$) and using this to establish interesting, although weak invariants, e.g., corresponding to our second requirement. The current development arose from different attempts to combine these developments and exploit the standard notions of convergence and deadlock-freeness as a way to express properties holding only in stable states.

Our different developments reflect not only the many facets of the problem, but also that there was a learning process involved in understanding the problem and its solution. The observation that this process is often nontrivial and requires iteration to converge on a good solution (and there may be many) is certainly not a new. But it is an observation worth repeating and such iteration fits well a development process where one alternates between specification and proving at different levels of abstraction.

References

1. Abrial, J.-R.: Modeling in Event-B: System and Software Design. Cambridge University Press, Cambridge (to appear, 2008)
2. Abrial, J.-R., Butler, M., Hallerstede, S., Voisin, L.: An open extensible tool environment for Event-B. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 588–605. Springer, Heidelberg (2006)
3. Abrial, J.-R., Cansell, D., Méry, D.: A mechanically proved and incremental development of IEEE 1394 tree identify protocol. Formal Asp. Comput. 14(3), 215–227 (2003)
4. Abrial, J.-R., Hallerstede, S.: Refinement, decomposition, and instantiation of discrete models: Application to Event-B. Fundamenta Informaticae, XXI (2006)
5. Back, R.-J., Kurki-Suonio, R.: Decentralization of process nets with centralized control. Distributed Computing 3(2), 73–87 (1989)
6. Bhargavan, K., Obradovic, D., Gunter, C.A.: Formal verification of standards for distance vector routing protocols. J. ACM 49(4), 538–576 (2002)
7. Clausen, T., Hansen, G., Christensen, L., Behrmann, G.: The Optimized Link State Routing Protocol, Evaluation through Experiments and Simulation. In: IEEE Symposium on Wireless Personal Mobile Communications (September 2001)
8. Clausen, T., Jacquet, P., Laouiti, A., et al.: Optimized Link State Routing Protocol. Request for Comments, 3626 (2003)
9. Dijkstra, E.W.: A note on two problems in connection with graphs. Numerische Mathematik 1, 269–271 (1959)
10. Hoang, T.S., Kuruma, H., Basin, D., Abrial, J.-R.: Developing topology discovery in Event-B. Technical Report 611, ETH Zurich, 11/2008
11. Lynch, N.: Distributed Algorithms. Morgan Kaufmann, San Francisco (1996)
12. Moy, J.T.: OSPF: Anatomy of an Internet Routing Protocol. Addison-Wesley Professional, Reading (1998)
13. Moy, J.T., et al.: OSPF Version 2 (1994)
14. Rfc3626: Optimized link state routing protocol (OLSR) (October 2003)
15. Udaya Shankar, A., Lam, S.S.: A stepwise refinement heuristic for protocol construction. ACM Transactions on Programming Languages and Systems 14(3), 417–461 (1992)
16. Tanenbaum, A.: Computer Networks. Prentice Hall Professional Technical Reference (2002)

Decomposition Structures for Event-B

Michael Butler

School of Electronics and Computer Science
University of Southampton, UK
mjb@ecs.soton.ac.uk

Abstract. Event-B provides a flexible approach to modelling and refinement of systems. In this paper we outline two important ways in which Event-B refinement can be augmented with additional structuring to support further the management of complex refinements. Firstly we show how event refinement diagrams can be used to structure refinement steps involving decomposition of atomicity. Secondly we outline a technique for decomposing models into sub-models to allow for independent refinement. We show how these two structuring techniques can be used together.

1 Introduction

An Event-B machine consists of a collection of variables, invariants on those variables and a collection of guarded events that may update the machine variables. An Event-B devlopment consists of a collection of machines linked by refinement.

Event-B [2] provides a more flexible approach to refinement than found in Classical B [1] and in related languages such as Z [10] and VDM [9]. One important feature is the ability to introduce new events in a refinement step. These new events correspond to stuttering steps that are not visible at an abstract level. A very common pattern of Event-B refinement for many types of system, including sequential, concurrent and distributed systems, is to represent a desired outcome as an abstract atomic event and then decompose that into smaller (sub-)atomic steps in refinement. While the Event-B refinement rules are quite comprehensive and allow for decomposition of event atomicity, they are more general than that. By identify a pattern and providing additional structure to represent the pattern, we hope to make the application of the standard refinement rules clearer and more manageable. In this paper we will see how a diagrammatic notation inspired by the structure diagrams of Jackson System Development (JSD) [8] can help to structure refinements involving atomicity decomposition.

Another critical structuring mechanism for refinement is the ability to decompose machines into sub-machines. Typically these sub-models will represent separate archtectural components. We will present a technique for syntactically partitioning an Event-B machine into several sub-machines. This technique has a sound semantic basis that corresponds to the synchronous parallel composition of processes as found in process algebra such as CSP [7]. An important property of the decomposition technique is that the resulting sub-models can be refined independently of each other.

2 Decomposing Atomicity

In this section we will look at how coarse-grained atomicity can be refined to more fine-grained atomicity. The approach we take is to treat most of the sub-atomic events of a decomposed abstract event as hidden events which are required to refine *skip*. The new events introduced in a refinement step can be viewed as hidden events not visible to the environment of a system and are thus outside the control of the environment. In Event-B, requiring a new event to refine *skip* corresponds to the process algebraic principle that the effect of an event is not observable. Any number of executions of an internal action may occur in between each execution of a visible action.

Assume we are refining a machine $M1$ by a machine $M2$. In Event-B, each event A of $M2$ either refines some event $R(A)$ of $M1$ or it is a new event refining *skip*. The proof obligations defined for Event-B refinement are based on the following proof rule that makes use of a gluing invariant J:

– Each $M2.A$ (data) refines $M1.R(A)$ under J, if $R(A)$ is defined
– Each $M2.A$ refines *skip* under J, if A is a new event

The machine L of Fig. 1 has a single event Out that simply outputs N and then disables itself. The machine contains a single variable for modelling the control of execution of the Out event: $Out \in BOOL$ is true when the output event has occurred. The Out event can occur provided Out has not occured ($grd1$). The parameter $v!$ represents the output value produced by the Out event. Its value is N ($grd2$).

We wish to refine this machine by a machine modelling a concurrent program that accumulates a value in a variable x before outputting it. The refinement

```
Machine L
Variables $Out$
Invariants $Out \in BOOL$
Initialisation $Out := FALSE$
Event Out $\widehat{=}$
    any $v!$ where
        $grd1 : Out = FALSE$
        $grd2 : v! = N$
    then
        $act1 : Out := TRUE$
    end
```

Fig. 1. Abstraction of model of simple outputting machine

models N parallel subprocesses each of which increments the variable x exactly once. When all N subprocesses have incremented x, the value of x is output. We view the refined model as breaking the atomicity of the output event by introducing an Inc event that models the behavior of the parallel sub-processes. The decomposition of the atomicity of the simple concurrent program is modelled as an *event refinement diagram* in Fig. 2. This diagrammatic notation is based on JSD structure diagrams by Jackson [8]. The event refinement diagram of Fig. 2 is a tree structure with root $Out(N)$ representing the abstract output event. The diagram shows how the root is decomposed into an initialisation, the parallel composition of multiple parallel instances of $Inc(p)$ and a refined output event $Out(x)$. The oval with the keyword **par** represents a quantifier that replicates the tree below it. In this case it replicates $Inc(p)$ by quantifying over sub-process identifiers p. An important feature of event refinement diagrams, in common with JSD structure diagrams, is that the subtrees are read from left to right and indicate sequential control from left to right. This means that our diagram indicates that the abstract $Out(N)$ event is realised in the refinement by firstly executing the initialisation, then executing the $Inc(p)$ events in parallel (in an interleaved fashion) and then executing $Out(x)$.

Another important feature of event refinement diagrams is the solid and dashed lines linking children to their parent. The $Init$ and $Inc(p)$ events are linked by a dashed line which means it must be proven that they refine $skip$. The abstract and refined Out events are linked by a solid line which indicates a refinement relation. That is, it must be proven that $Out(x)$ refines $Out(N)$.

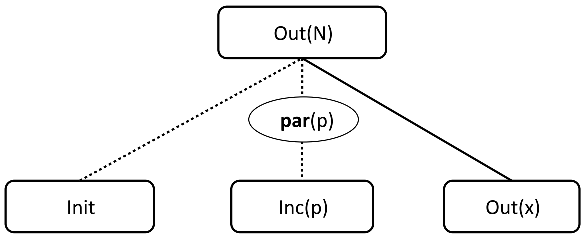


Fig. 2. Event refinement diagram illustrating atomicity decomposition

The refined machine is shown in Fig. 3. It uses a type $PROC$ representing the set of sub-process identifiers with the assumption that $card(PROC) = N$. In addition to the variable x, machine M contains two variables for modelling the control of execution of events. Variable $Inc \subseteq PROC$ represents the set of processes for which the increment event has occurred. Variable $Out \in BOOL$ is true when the output event has occurred. In this case the initialisation of the program is modelled by the standard initialisation clause of the machine M so we do not need a control variable for the initialisation. The Inc event can occur for process p provided Inc has not already occurred for process p. This constraint is modelled by guard $grd1$ of Inc. The action $act1$ of the Inc event

```
Machine  M
Variables   $x$, $Inc$, $Out$
Invariants   $x \in \mathbb{N}$,  $Inc \subseteq PROC$,  $Out \in BOOL$
Initialisation   $x := 0$,  $Inc := \{\}$,  $Out := FALSE$
Event  Inc $\widehat{=}$
    any $p$ where
        $grd1 : p \in PROC \setminus Inc$
    then
        $act1 : Inc := Inc \cup \{p\}$
        $act2 : x := x + 1$
    end
Event  Out $\widehat{=}$
    any $v!$ where
        $grd1 : Inc = PROC$
        $grd2 : Out = FALSE$
        $grd3 : v! = x$
    then
        $act1 : Out := TRUE$
    end
```

Fig. 3. Event-B refinement of a simple output machine

adds the value p to the set Inc which prevents the event occurring for that value of p again. The Out event can occur provided Inc has occurred for all processes ($grd1$) and Out has not occured ($grd2$).

Instead of outputting N the refined Out event outputs the value of x ($grd3$). The proof of the correctness of this refinement relies on the following invariant stating that the value of x is equal to the number of processes that have completed their task:

$$x = card(Inc)$$

Therefore when all N processes have completed, x will have the value N and the correct value will be output. This illustrates how control variables (such as Inc) are useful in gluing invariants, allowing for values of data variables (such as x) to be related to values of control variables.

Consider the case where we have two subprocesses so that $PROC = \{p1, p2\}$ and $N = 2$. The event traces of the model are as follows:

$$\langle\ Inc.p1,\ Inc.p2,\ Out.2\ \rangle \qquad \langle\ Inc.p2,\ Inc.p1,\ Out.2\ \rangle$$

Each event trace represents a record of a possible execution trace of the model. Here we are ignoring the initialisation event since it always occurs exactly once at the beginning of a trace. The parallel execution of the subprocesses is modelled by interleavings of the atomic steps of the processes. Here the two possible interleavings of $Inc.p1$ and $Inc.p2$, represented by the two events traces, model their concurrent execution. It is instructive to relate the event traces of the machine L with those of machine M. L has just a single event trace that outputs N and nothing else. In the case that $N = 2$, the single event trace of L is

$$\langle\ Out.2\ \rangle$$

If we remove the Inc events from the traces of M we get the trace of L:

$$\begin{array}{lcl}\langle\ Inc.p1,\ Inc.p2,\ Out.2\ \rangle \setminus Inc & = & \langle\ Out.2\ \rangle \\ \langle\ Inc.p2,\ Inc.p1,\ Out.2\ \rangle \setminus Inc & = & \langle\ Out.2\ \rangle\end{array}$$

Removing events from a trace is the standard way of giving a semantics to hidden or stuttering events and is used, for example, in CSP. By treating the Inc events as a hidden, traces of M look like traces of L. This illustrates a semantics of refinement of Event-B models. Machine M is a refinement of machine L since any trace of M in which the Inc events are hidden is also a trace of L. This is treated more precisely in [5].

3 Decomposing File Write

We will study a further example of atomicity refinement which involves more event interleaving than the simple concurrent program. This is an event for writing a file to a disk. At the abstract level the entire contents of the file is written in one atomic step as in the following machine:

```
Machine  File1
Variables   $file,\ dsk$
Invariants   $file \subseteq FILE$,   $dsk \in file \rightarrow CONT$

Event  Write $\widehat{=}$
    any  $f, c$  where
        $grd1:\ f \in file$
        $grd2:\ c \in CONT$
    then
        $act1:\ dsk(f) := c$
    end
```

Here the contents of the disk are represented by the variable dsk which maps file identifiers to their contents. The $Write$ event has two parameters, the identity of the file to be written f and the contents to be written c. Other events such as creating a file and reading a file are not shown.

We assume that file contents are structured as a set of pages of data so that the type $CONT$ is defined as follows:

$$CONT = PAGE \leftrightarrow DATA$$

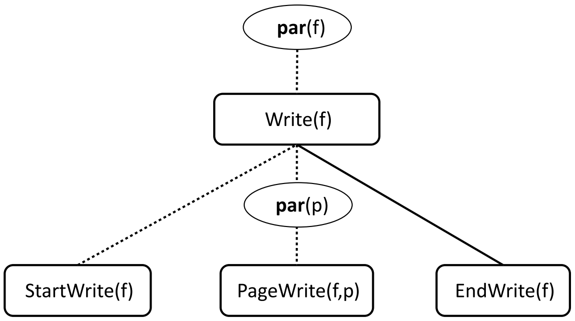


Fig. 4. Decomposition of the atomicity of file write

The event refinement diagram of Fig. 4 illustrates the decomposition of the $Write$ event into sub-events to model the writing of individual pages. In the refinement, the writing of individual pages will be modelled atomically by the $PageWrite$ event and the writing of the entire file is no longer atomic. The writing of a file is initiated by the $StartWrite$ event and ended by the $EndWrite$ event. We will allow multiple file writes to be taking place simultaneously in an interleaved fashion. This is indicated by the top level parallel quantification over f ($\mathbf{par}(f)$). We also assume that the pages of an individual file f can be written in parallel hence the inner parallel quantification over p ($\mathbf{par}(p)$). Occurrence of event $PageWrite(f, p)$ models writing of page p of file f.

In order to model the event sequencing implied by Fig. 4, we introduce variables corresponding to the $StartWrite$ and $PageWrite$ events as follows:

Invariants
$inv1: StartWrite \subseteq FILE$
$inv2: PageWrite \subseteq FILE \times PAGE$
$inv3: dom(PageWrite) \subseteq StartWrite$

The types of these variables are determined by the parallel quantification in Fig. 4. $StartWrite$ is a subset of $FILE$ because it is bound by the quantification over files f ($inv1$). $PageWrite$ is a subset of $FILE \times PAGE$ because it is bound by the quantification over files f and pages p ($inv2$). If a page has been written for a file, then $StartWrite$ will already have occurred for that file ($inv3$).

When the writing of a file is complete, we will allow the file to be written to again. Therefore we do not need any variable to model the occurrence of the $EndWrite$ event for a file, since all the control information for a file will be cleared when the file write is complete in order to allow the file to be written to again later if required. Now, for example, the control behaviour of the $StartWrite$ and $PageWrite$ events is as follows:

Event StartWrite $\widehat{=}$
any f **where**
$grd1: f \in file$
$grd2: f \notin StartWrite$

```
    then
        $act1 : StartWrite := StartWrite \cup \{f\}$
    end
Event PageWrite $\widehat{=}$
    any $f, p$ where
        $grd1 : f \in StartWrite$
        $grd2 : f \mapsto p \notin PageWrite$
    then
        $act1 : PageWrite := PageWrite \cup \{f \mapsto p\}$
    end
```

This control behaviour on its own is not enough. The pages and their contents for a particular file need to be determined before we start the process of writing to a file. We introduce a variable *writebuf* to act as a buffer for the content to be written to disk. Rather than writing directly to the abstract variable *dsk*, the *PageWrite* event will write the contents of an indivdual page to a shadow disk while the writing is in progress. When the writing is complete, the contents of the shadow disk is transferred to the disk at the end of the writing process. These variables are defined as follows:

$$inv4 : writebuf \in StartWrite \to CONT$$
$$inv5 : sdsk \in StartWrite \to CONT$$

Note that both are defined on files that are currently being written, i.e., files in the set *StartWrite*.

Now, as well as initialising the control for the writing process, the *StartWrite* event sets the contents to be written to disk in the write buffer for that file ($act2$) and sets the shadow disk for that file to be empty ($act3$):

```
Event StartWrite $\widehat{=}$
    any $f, c$ where
        $grd1 : f \in file$
        $grd2 : f \notin StartWrite$
        $grd3 : c \in CONT$
    then
        $act1 : StartWrite := StartWrite \cup \{f\}$
        $act2 : writebuf(f) := c$
        $act3 : sdsk(f) := \varnothing$
    end
```

The *PageWrite* event selects a page of a file that has yet to be written ($grd2$) and is in the write buffer ($grd3$). The parameter d represents the data associated with the page being written to the shadow disk (*sdsk*):

```
Event PageWrite $\widehat{=}$
    any $f, p, d$ where
        $grd1 : f \in StartWrite$
        $grd2 : f \mapsto p \notin PageWrite$
        $grd3 : p \mapsto d \in writebuf(f)$
```

```
then
    $act1 :\ PageWrite := PageWrite \cup \{f \mapsto p\}$
    $act2 :\ sdsk(f) := sdsk(f) \Leftarrow \{p \mapsto d\}$
end
```

The $StartWrite$ and $PageWrite$ events both refine $skip$ while the $EndWrite$ event refines the abstract $Write$ event (see the dashed and solid lines in Fig. 4). The $EndWrite$ event occurs once all pages of a file have been written, a condition that is captured by $grd2$ below. The effect of the event is to copy the shadow disk to the disk ($act1$). The event also clears all the control, buffer and shadow information for the file to enable the write process to commence all over again ($act2$ to $act5$).

```
Event  EndWrite  Refines  Write  $\widehat{=}$
    any  $f, c$  where
        $grd1 :\ f \in StartWrite$
        $grd2 :\ PageWrite[\{f\}] = dom(writebuf(f))$
        $grd3 :\ c = sdsk(f)$
    then
        $act1 :\ dsk(f) := sdsk(f)$
        $act2 :\ StartWrite := StartWrite \setminus \{f\}$
        $act3 :\ PageWrite := \{f\} \lhd PageWrite$
        $act4 :\ writebuf := \{f\} \lhd writebuf$
        $act5 :\ sdsk := \{f\} \lhd sdsk$
    end
```

It may seem like we have not really achieved much decomposition of atomicity since the shadow disk is copied to the actual disk in one atomic step ($act1$ of $EndWrite$). However our intention is that the disk and the shadow together are both realised on the real hard disk and that the effect of $act1$ would be achieved by an update to the page table for the disk (in later refinements). We assume that updating the page table can reasonably be treated as atomic. Having the $PageWrite$ event write the individual pages to a shadow disk also allows us to model fault tolerance quite easily. We add an $AbortWrite$ event that clears all the control and shadow information for a file write but does not update the disk:

```
Event  AbortWrite $\widehat{=}$
    any  $f$  where
        $grd1 :\ f \in StartWrite$
    then
        $act1 :\ StartWrite := StartWrite \setminus \{f\}$
        $act2 :\ writebuf := \{f\} \lhd writebuf$
        $act3 :\ sdsk := \{f\} \lhd sdsk$
        $act4 :\ PageWrite := \{f\} \lhd PageWrite$
    end
```

This event refines *skip* since it does not modify the *dsk* variable that appears in the abstract model. Thus the effect of an abort, which can happen after any number of pages are written, is to leave the disk in the state it was in before the file write process started (for the file f).

It is instructive to compare an event trace of the abstract file model with a corresponding trace of the refinement file model. The following trace represents a behaviour in which the contents $c2$ is written to file $f2$ and then the contents $c1$ is written to file $f1$:

$$\langle\ \mathit{Write}.f2.c2,\ \ \mathit{Write}.f1.c1\ \rangle$$

Each of these high-level events is realised by several new events (*StartWrite*, *PageWrite* etc). The sub-events of one high-level write may interleave with those of the other high-level event. For example, the following event trace of the refined model illustrates this (the events that directly refine an abstract event are highlighted in bold):

$$\begin{array}{l}\langle\ \mathit{StartWrite}.f1.c1,\ \ \mathit{PageWrite}.f1.p1.c1(p1),\\ \mathit{StartWrite}.f2.c2,\ \ \mathit{PageWrite}.f1.p2.c1(p2),\\ \mathit{PageWrite}.f2.p1.c2(p1),\ \ \mathit{PageWrite}.f2.p2.c2(p2),\\ \mathbf{EndWrite}.\mathbf{f2}.\mathbf{c2},\ \ \mathit{PageWrite}.f1.p3.c1(p3),\ \ \mathbf{EndWrite}.\mathbf{f1}.\mathbf{c1}\ \rangle\end{array}$$

This illustrates a scenario in which writing to file $f1$ is started before writing to $f2$ is started but writing of file $f2$ finishes before writing of file $f1$.

To recap, we have decomposed the atomicity of the abstract *Write* event by introducing the new events *StartWrite*, *PageWrite* and *AbortWrite* and by refining the *Write* event with the *EndWrite* event. Formally, the new events have no connection to the abstract *Write* event, only the *EndWrite* has a formal connection. However, the event refinement diagram of Fig. 4 describes the intended purpose of the new events which is to represent the intermediate steps of the file write process that lead to a state where the *EndWrite* is enabled. The diagram also plays another role in that it defines the control behaviour of all the events constituting the write process and this was encoded in Event-B in a systematic way, i.e., introducing the *StartWrite* and *PageWrite* control variables. The additional modelling elements provided, *writebuf* and *sdsk*, were required in order to model abstractly the effect of the various events and their introduction was based on modelling judgement.

4 Decomposing Machines

In this section, we describe a parallel composition operator for machines. The parallel composition of machines M and N is written $M \parallel N$. Machines M and N must not have any common state variables. Instead they interact by synchronising over shared events (i.e., events with common names). They may also pass values on synchronisation. We look first at basic parallel composition and later look at parallel composition with shared parameters. We show how the composition operator may be applied in reverse in order to decompose system models into subsystem models.

In general, an event has the form

$$\textbf{any } x \textbf{ where } G \textbf{ then } S \textbf{ end}$$

where x is a list of event parameters, G is a list of guards (implicitly conjoined) and S is a list of actions on the machine variables (implicitly simultaneous). We write $G \wedge H$ to join two lists of guards and $S \parallel T$ to join two lists of actions.

To achieve the synchronisation effect between machines, shared events from M and N are 'fused' using a parallel operator for events. Assume that m (resp. n) represents the state variables of machine M (resp. N). Variables m and n are disjoint. The parallel operator for events is defined as follows:

$$\begin{array}{lcl} ev1 & = & \textbf{any } y \textbf{ where } G(y,m) \textbf{ then } S(y,m) \textbf{ end} \\ ev2 & = & \textbf{any } z \textbf{ where } H(z,n) \textbf{ then } T(z,n) \textbf{ end} \end{array}$$

$$\begin{array}{lcl} ev1 \parallel ev2 & \widehat{=} & \textbf{any} \quad y, z \quad \textbf{where} \\ & & \quad G(y,m) \wedge H(z,n) \\ & & \textbf{then} \\ & & \quad S(y,m) \parallel T(z,n) \\ & & \textbf{end} \end{array}$$

The parallel operator models simultaneous execution of the actions of the events and the composite event is enabled exactly when both component events are enabled. This models synchronisation: the composite system engages in a joint event when both systems are willing to engage in that event. The parallel composition of machines M and N is a machine constructed by fusing shared events of M and N and leaving independent events independent. The state variables of the composite system $M \parallel N$ are simply the union of the variables of M and N.

As an illustration of this, consider machines *V1* and *W1* of Fig. 5. The machines work on independent variables v and w respectively. Both machines have an event labelled B and to compose these machines we fuse their respective B events. The composition of both machines is shown in Fig. 6. The A event and C event of *VW1* come directly from *V1* and *W1* respectively as they are not joint events rather they are independent events. The B event is a joint event and is defined as the fusion of the B-events of *V1* and *W2*. The initialisations of *V1* and *W1* are also combined to form the initialisation of *VW1*. The joint B event simultaneously decreases v while increasing w, provided $v > 0$ and $w < N$.

We have presented *VW1* as having been formed from the composition of *V1* and *W1*. We can view the relationship between these machines in another way. Let us suppose we had started with *VW1* and decided that we wish to decompose

```
Machine  V1
Variables  v
Invariants  $v \in \mathbb{N}$
Initialisation  $v := N$
Event  B $\widehat{=}$
    when
        $grd1 :\ v > 0$
    then
        $act1 :\ v\ :=\ v - 1$
    end
Event  A $\widehat{=}$
    begin
        $act1 :\ v\ :=\ N$
    end
```

(a) Machine $V1$

```
Machine  W1
Variables  w
Invariants  $w \in \mathbb{N}$
Initialisation  $w := 0$
Event  B $\widehat{=}$
    when
        $grd2 :\ w < M$
    then
        $act2 :\ w\ :=\ w + 1$
    end
Event  C $\widehat{=}$
    when
        $grd1 :\ w > 0$
    then
        $act1 :\ w\ :=\ w - 1$
    end
```

(b) Machine $W1$

Fig. 5. Machines to be composed in parallel

it into subsystems. The diagram in Fig. 7(a) illustrates the dependencies between events and variables in the machine *VW1*. For example, the line from the box indicating event A to the oval indicating variable v represents the fact that event A depends on v, i.e., it may read from and assign to v. The diagram shows that B is the only event that depends on both v and w suggesting that B needs to be a shared event if we are to partition v and w into separate subsystems. This decomposition is illustrated in Fig. 7(b) where variables v and w of *VW1* are partitioned into subsystems *V1* and *W1* respectively, A is an event of subsystem *V1*, C is an event of subsystem *W1* and B is an event shared by both subsystems.

The B event of system *VW1* is partitioned into two parts, one of which will belong in *W1* and the other in *W1*. The B event has an important characteristic that allows it to be partitioned in this way. The guards and actions depend either on v or on w but not both. So, guard $grd1$ and action $act1$ both depend on v only, while guard $grd2$ and action $act2$ both depend on w. This localisation of variable dependency allows us to easily partition the guards and actions of the B event of *VW1* into the separate B events of *V1* and *W1* respectively.

We extend the fusion operator to deal with shared event parameters. Events to be fused must depend on disjoint machine variables but they may have common parameters and these common parameters are treated as joint parameters in the fused event. In the following, x represents parameters that are joint across events and y and z are local to their respective events:

$$\begin{aligned} ev1 &\quad=\quad \textbf{any}\ x, y\ \textbf{where}\ G(x,y,m)\ \textbf{then}\ S(x,y,m)\ \textbf{end} \\ ev2 &\quad=\quad \textbf{any}\ x, z\ \textbf{where}\ H(x,z,n)\ \textbf{then}\ T(x,z,n)\ \textbf{end} \end{aligned}$$

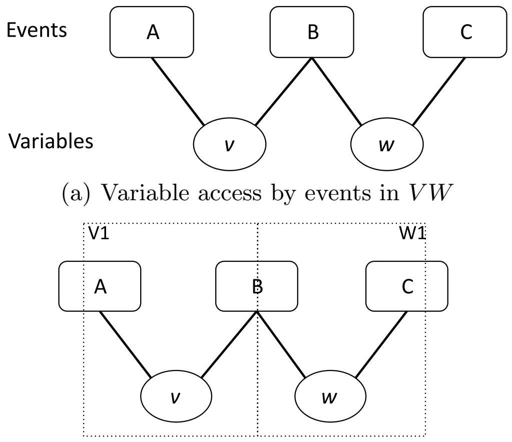
```
Machine VW1
Variables  v,  w
Invariants  $v \in \mathbb{N}$,  $w \in \mathbb{N}$
Initialisation  $v := N$,  $w := 0$
Event  A $\widehat{=}$
    begin
        $act1 :\ v\ :=\ N$
    end
Event  B $\widehat{=}$
    when
        $grd1 :\ v > 0$
        $grd2 :\ w < M$
    then
        $act1 :\ v\ :=\ v - 1$
        $act2 :\ w\ :=\ w + 1$
    end
Event  C $\widehat{=}$
    when
        $grd1 :\ w > 0$
    then
        $act1 :\ w\ :=\ w - 1$
    end
```

Fig. 6. Composition of $V1$ and $V2$

```
$ev1 \parallel ev2$  $\widehat{=}$  any  $x, y, z$  where
                          $G(x, y, m) \wedge H(x, z, n)$
                      then
                          $S(x, y, m) \parallel T(x, z, n)$
                      end
```

We illustrate the use of shared parameters by extending the *VW1* machine slightly. Assume that instead of increasing v and decreasing w by 1 in the B event, we modify both v and w by a value i. To do this we give the B event a parameter i which is used to modify the variables as follows:

```
Event  B $\widehat{=}$
    any  i  where
        $grd1 :\ 0 \leq i \leq v$
        $grd2 :\ w < N$
    then
        $act1 :\ v\ :=\ v - i$
        $act2 :\ w\ :=\ w + i$
    end
```



(a) Variable access by events in VW

(b) Split events and variables

Fig. 7. Illustration of decomposition a machine

Now we partition the guards and events of B into those that depend on v and those that depend on w giving the following events:

```
Event  B ≙
    any  i  where
        $grd1 : 0 \leq i \leq v$
    then
        $act1 : v := v - i$
    end
```

```
Event  B ≙
    any  i  where
        $grd1 : i \in \mathbb{Z}$
        $grd2 : w < N$
    then
        $act1 : w := w + i$
    end
```

The shared parameter i means that both of these events will agree on the amount by which v and w are respectively decreased and increased. In the left hand sub-event, the guard $grd1$ constraints the value of the parameter based in the state variable v. In the right-hand sub-event, the value of i is not constrained other than a typing guard ($i \in \mathbb{Z}$). This means that the left-hand sub-event can be viewed as outputting the value i while the right-hand sub-event accepts the value i as an input.

When we decompose a system into parallel subsystems, the subsystems may be refined and further decomposed independently. This is a major methodological benefit, helping to modularise the design and proof effort. The semantic justification for this is outlined in [5].

5 Incremental Development of a Distributed File Transfer

In this section we outline an incremental development of a simple system for copying a file from one location to another. The development makes use of event

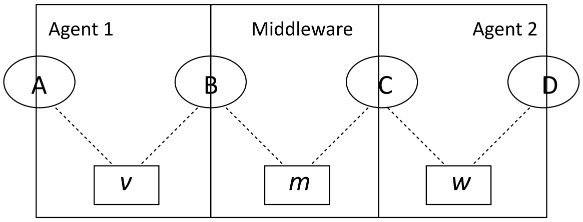


Fig. 8. Decomposition with asynchronous middleware

decomposition and machine decomposition. We start with an abstract model in which the file copy occurs in one atomic step. We then refine this by a model in which the contents of the file is copied one page at a time. The refined model is then decomposed into subsystems. Instead of decomposing into two subsystems that synchronise with each other, we decompose into three subsystems as illustrated in Fig. 8. In this decomposition the two agents do not synchronise directly with each other. Instead they interact indirectly through a middleware subsystem. Each agent synchronises directly and separately with the middleware and this will be used to model asynchronous communication between the agents. This form of asynchronous communication via middleware can be used to model many distributed systems that are based on message passing. In order to be able to decompose in this way, we will need to apply refinement steps that enable the agents to be decomposed into asynchronous subsystems.

5.1 Abstract Model

The model makes use of the types $PAGE$ and $DATA$ respectively. A file is modelled as a partial function from pages to data. Machine $F1$ defines the abstract behaviour of the file transfer system. It contains two variables $fileA$, representing the contents of the file at the sending side, and $fileB$ representing the value of the file at the receiving side:

Machine F1
Variables *fileA* , *fileB*
Invariants
 $inv1 : \ fileA \in PAGE \rightarrowtail\mkern-14mu\mapsto DATA$
 $inv2 : \ fileB \in PAGE \rightarrowtail\mkern-14mu\mapsto DATA$

The abstract machine has one event that simply copies the contents of $fileA$ to $fileB$ in one atomic step:

```
Event CopyFile ≙
    begin
        $act1 : \ fileB := fileA$
    end
```

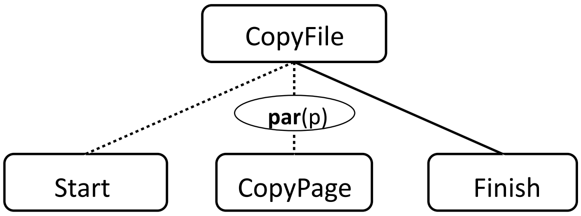


Fig. 9. Refining atomicity of the *CopyFile* event

5.2 Breaking Atomicity

The atomicity of the *CopyFile* event is decomposed in the same way in which the atomicity of the *Write* event was decomposed in Section 2. This is illustrated in Fig. 9. We introduce control variables based on this diagram as well as a buffer *buf* in which pages are written one at a time by the *CopyPage* event. Further details of this refinement may be found in [5].

5.3 Split Events to A Side and B Side

Before decomposing the file transfer system into three subsystems, we must first split some events into an A-part, representing behaviour on the sending side, and a B-part, representing behaviour on the receiving side. This is illustrated by the diagram in Fig. 10 which shows that the *Start* event is decomposed into *StartA* and *StartB*. The *StartA* event represents the sending side deciding to commence the transfer while the subsequent *StartB* event represents the receiving side recognising that the transfer has commenced. The *StartA* event will set a flag *StartA* to *TRUE* while the *StartB* event will set a flag *StartB* to *TRUE* provided *StartA* is true. The *CopyPage* event is decomposed into separate A and B parts in a similar way. We assume that the sending side will send the size of the file at the start so that the receiving side can know when all the pages have been received. This means that the sending side does not need to send a finish message so we need a *Finish* event on the receiving side only.

The event refinement diagram in Fig. 10 provides a hierarchical overview of the major refinement steps involved in this development so far. The top level corresponds to the abstract atomic event, the intermediate level corresponds to the first refinement where the atomicity of the copy is decomposed and the third level of the hierarchy shows how events are split into two parts for sender and receiver.

5.4 Introduce Message Variables

Now consider again the *StartB* event just outlined. Our intention is that this is an event of the receiving side so we wish to make it an event of the receiver subsystem. This means it should not refer to variables of the sending side directly since we are aiming at an asynchronous decomposition. However the *StartB*

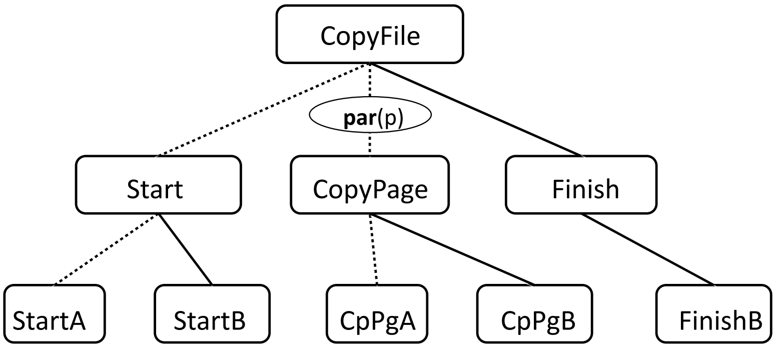


Fig. 10. Splitting events into sender and receiver parts

event does refer to variables of the sending side: for example it refers to the *StartA* control variable.

To break this dependency on variables of the sending side in events of the receiving side, we introduce variables that duplicate the variables of the sending side, e.g., *StartM* and *CopyPageM*. These duplicate variables will be separated into a middleware machine (Fig. 8) and become abstract representations of messages in transit in the middleware.

5.5 Separate Machines

The previous model is decomposed into three separate machines representing three subsystems as illustrated in Fig. 8. The three machines are:

- machine $mA1$ representing a model of the sending agent
- machine $mB1$ representing a model of the receiving agent
- machine $mM1$ representing a model of the middleware through which the sender and receiver interact.

The variables of the previous model are partitioned amongst the three machines. The sender interacts with the middleware through synchronisation over actions (*StartA* and *CopyPageA*). Similarly, the receiver interacts with the middleware through synchronisation over actions (*StartB* and *CopyPageB*). There is no direct interaction between the sender and receiver - all communication is via the middleware machine.

Fig. 11 provides an architectural overview of the decomposition illustrating how the variables and events are distributed amongst the subsystems. The variables allocated to each subsystem are listed in italic in the relevant box for that subsystem, e.g., the sender subsystem contains the variables *fileA*, *StartA* etc. The smaller labelled boxes indicate the synchronised shared events. For example, the *StartA* event is shared between the sender and the middleware representing a synchronised interaction between these subsystems.

See [5] for further details of how the event specifications are decomposed into the separate syntactic components in order to decompose the model. [5] also

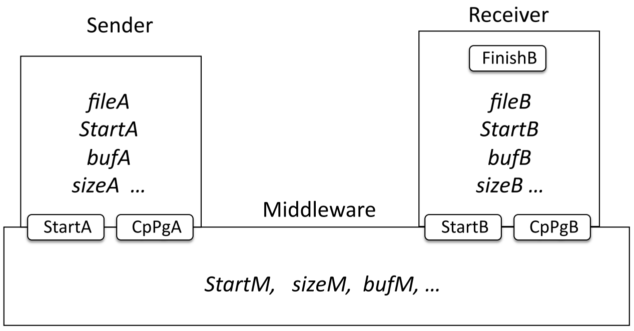


Fig. 11. Architectural illustration of decomposition

outlines how the abstract model of the middleware may be refined further so that more explicit datatypes representing messages are introduced reflecting the usual interface to a communications middleware.

6 More about Event Refinement Diagrams

In the event refinement diagrams shown so far, the refining event is always the final step of an event decomposition. For example, in Fig. 2, the refined $Out(N)$ event is the final step in the decomposition of the abstract $Out(N)$ event. It is not a requirement that the refining event always be the final event of a decomposition. Fig. 12 shows an event refinement diagram for an update of a replicated database in which the refining event is followed by further new events. This diagram is based on the structure of a refinement presented in [11] (although event refinement diagrams are not used in [11]). The outline of this development is as follows. The abstract machine models a single database. The refined machine models a set of sites each of which holds its own copy of the database. In the abstract machine, an update of the database is a simple atomic event. The refinement uses a two-phase commit protocol (with *precommit* then *commit* phases) to ensure a consistent distributed update transaction. The phasing is represented in Fig. 12. Once an update transaction t is started, each site s independently precommits to the transaction (which locks all the database objects involved in the transaction). Once all sites have precommitted, the transaction is globally committed by a coordinator. The *GlobalCommit* refines the abstract *Update* since a global decision has been made to update all copies of the database. After the global commit, each site s locally commits its copy of the database independently (and releases any objects locked by its precommit).

In this paper we have avoided providing a systematic definition of event refinement diagrams and their translation to Event-B. The reason for this is simply that the concepts are not fully mature at the time of writing. It may be that a complete set of translation rules is not appropriate and that instead a common set of patterns can be identified and translations provided for those. The diagrams seem to be a promising way of representing reusable patterns of event

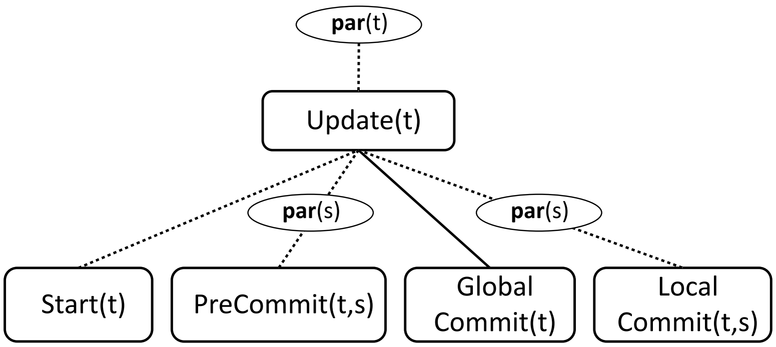


Fig. 12. Event refinement diagram for replicated database update

decomposition. They are abstract and visual and humans are good are recognising visual patterns. This is one reason why we have avoided cluttering the diagrams too much with, for example, event guard. Too much clutter may make patterns appear less general.

Our initial exploration of JSD structure diagrams as a means of representing the structure of atomicity decomposition was influenced by the work of Ball [4] on the use of KAOS [6] goal diagrams for a similar purpose. Our event refinement diagrams are different in construction to the refinement diagrams developed by Back [3]. Back's diagrams expose the containment and refinement relationships between general components and subcomponents. In Back's diagrams, enclosing components may be replicated in order to simultaneously illustrate refinements between subcomponents and between enclosing components. In our diagrams the higher level events can be viewed as enclosing components and these only appear once at the top level. Back's diagrams are neutral with respect to the operator used to compose components. In our diagrams the operators (sequential and parallel) are built in.

7 Concluding

We have outlined techniques for atomicity decomposition and machine decomposition. The atomicity decomposition technique uses the standard Event-B refinement rule together with event refinement diagrams to provide an explicit representation of the the sequencing of sub-events and the refinement relationships involved. These diagrams provide a systematic means of introducing control structure in an incremental manner through diagram hierarchy. They provide a useful hierarchical overview of multiple refinement steps. They provide a convenient mechanism for exploring several levels of event decomposition in advance of construction of the appropriate Event-B refinements. They also appear to provide a convenient way of representing reusable patterns of event refinements. The machine decomposition technique is based on synchronisation between machines over shared events with asynchronous decomposition as a special case involving an explicit representation of an asynchronous communications medium. The decomposition approach supports independent refinement and decomposition of sub-machines. Together, the event decomposition and machine decomposition techniques augment Event-B by making the application of refinement more systematic and scalable then the standard refinement rules on their own.

Acknowledgements. The work described here is part of the EU research project ICT 214158 DEPLOY (Industrial deployment of system engineering methods providing high dependability and productivity) www.deploy-project.eu.

References

1. Abrial, J.-R.: The B-Book: Assigning programs to meanings. Cambridge University Press, Cambridge (1996)
2. Abrial, J.-R.: Modelling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2008)
3. Back, R.-J.: Refinement diagrams. In: Morris, J.M., Shaw, R.C. (eds.) Proceedings of the 4th Refinement Workshop, Cambridge, UK, Jan 1991, pp. 125–137. Springer, Heidelberg (1991)
4. Ball, E.: An Incremental Process for the Development of Multi-agent Systems in Event-B, PhD thesis, University of Southampton (August 2008), `http://eprints.ecs.soton.ac.uk/16575/`
5. Butler, M.: Incremental design of distributed systems with Event-B. Marktoberdorf Summer School 2008 Lecture Notes (November 2008)
6. Dardenne, A., van Lamsweerde, A., Fickas, S.: Goal-directed requirements acquisition. Sci. Comput. Program. 20(1-2), 3–50 (1993)
7. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
8. Jackson, M.A.: System Development. Prentice-Hall, Englewood Cliffs (1983)
9. Jones, C.B.: Systematic Software Development using VDM, 2nd edn. Prentice Hall International, Englewood Cliffs (1990)
10. Spivey, J.M.: The Z Notation: A Reference Manual, 2nd edn. Prentice Hall International Series in Computer Science (1992)
11. Yadav, D.S., Butler, M.J.: Formal development of fault tolerant transactions for a replicated database using ordered broadcasts. In: Methods, Models and Tools for Fault Tolerance (MeMoT 2007), May 2007, pp. 33–42 (2007)

Taming the Unbounded for Hardware Synthesis

Byron Cook

Microsoft Research Cambridge

Abstract. The difficulty with compiling software to hardware is one of finding *a priori* bounds on the potentially unbounded resources used by programs: memory and time. New approaches now allow us to synthesize bounds on these resources, leading to new high-level hardware synthesis tools.

Verifying UML/OCL Operation Contracts

Jordi Cabot[⋆], Robert Clarisó, and Daniel Riera

Universitat Oberta de Catalunya, Spain
{jcabot,rclariso,drierat}@uoc.edu

Abstract. In current model-driven development approaches, software models are the primary artifacts of the development process. Therefore, assessment of their correctness is a key issue to ensure the quality of the final application. Research on model consistency has focused mostly on the models' static aspects. Instead, this paper addresses the verification of their dynamic aspects, expressed as a set of operations defined by means of pre/postcondition contracts.

This paper presents an automatic method based on Constraint Programming to verify UML models extended with OCL constraints and operation contracts. In our approach, both static and dynamic aspects are translated into a Constraint Satisfaction Problem. Then, compliance of the operations with respect to several correctness properties such as operation executability or determinism are formally verified.

1 Introduction

In recent years, Model Driven Development (MDD) is gaining attention due to its promise to increase productivity in developing, documenting, and maintaining software systems. MDD emphasizes the use of models during the whole development process and thus the *correctness of a model* becomes a major issue: model defects will directly become implementation defects in the final software system due to the application of code-generation techniques. Unfortunately, popular modeling notations (UML [5] being the most widely used) are not formal enough to directly prove the correctness of the software models. Therefore, a set of model-level verification techniques are needed to ensure the quality of software model specifications. Each technique can address a variety of correctness properties and goals depending on which type of models it is analyzing.

In particular, this paper presents a new method for the verification of the behavioural aspects of software models defined using the design by contract approach [20], where each operation is defined by means of a contract consisting of a *precondition* (set of conditions on the operation input) and a *postcondition* (conditions to be satisfied at the end of the operation). In conceptual modeling, this is also known as the declarative specification of an operation, in contrast to imperative specifications where the set of updates produced by the operation on the system state is explicitly defined. Our goal will be detecting defects in

⋆ Work partly supported by the Ministerio de Educación y Ciencia, FEDER under project TIN208-00444/TIN, Grupo Consolidado and UOC-IN3 research grant.

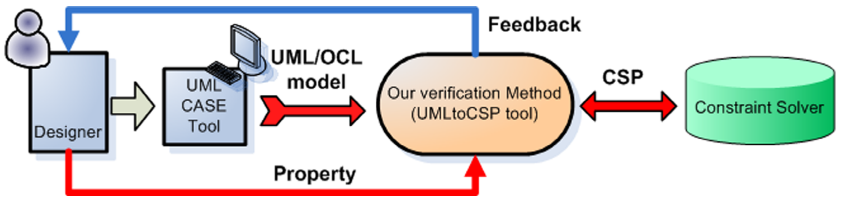


Fig. 1. Overall picture of the process

the definition of the operation (e.g. potential inconsistent interactions with integrity constraints) rather than checking whether an implementation fulfills the pre/postconditions. This is an extension of our previous work [9,19] which focused only on reasoning on integrity constraints without considering operations.

The goal of this paper is twofold. First, we present a set of "reasonable" correctness criteria that any operation should fulfill. For example, we will try to check if a precondition is so strong that it cannot be satisfied by any state that fulfills the integrity constraints (e.g. a precondition "a $\geq$ 5" when the model includes the constraint "a $\leq$ 3" is clearly unsatisfiable). Designers can select their preferred set of criteria among the predefined set of properties we propose.

Second, we provide a method for verifying these properties on UML/OCL models. Without loss of generality, we will assume that our input model is a UML class diagram, annotated with integrity constraints, and pre/postconditions written in the Object Constraint Language (OCL) [15]. Our choice is based on the wide adoption of the UML and it high-level modeling constructs, although many concepts of this work are applicable to other modeling languages as well.

The verification will be driven by the discovery of examples/counterexamples. First, the designer selects the criteria to be checked. The model, the integrity constraints, the correctness criteria and the pre/postconditions will be transformed into a Constraint Satisfaction Problem (CSP) [2,14] that can be solved by current Constraint Programming solvers. The solution of the CSP, if there is one, will be an example or counterexample that proves the criteria being analyzed. The example is given to the designer as a valuable feedback in the form of an object diagram (so that he/she can understand it). Our UMLtoCSP tool [19] will be used to automate the process (see Figure 1).

The rest of the paper is structured as follows. Section 2 introduces OCL concepts. Section 3 presents our correctness properties and Section 4 their verification with constraint programming. Tool support is commented in section 5. Section 6 discusses related work and Section 7 draws some conclusions.

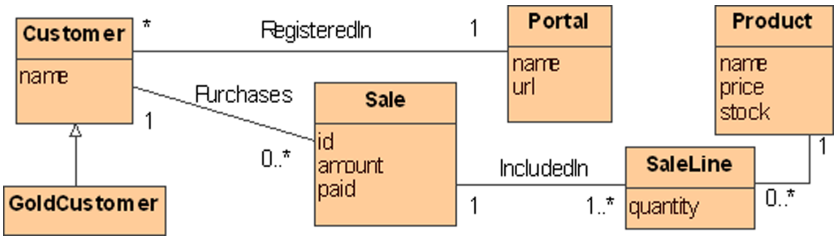
2 Declarative Operations in OCL: Basic Concepts

OCL is a formal high-level language used to describe properties on UML models. It admits several powerful constructs like quantified iterators (*forAll*, *exists*) and operations over collections of objects (*union*, *select*, *includes*, . . .). The pattern for specifying a declarative operation *op* in OCL is the following:

context TypeName::op(p1: Type1, ..., pN: TypeN): ResultType
pre: Boolean expression (the precondition)
post: Boolean expression (the postcondition)

Operations are always defined in the context of a specific type of the model. The *pre* and *post* clauses are used to express the preconditions and postconditions of the operation contract. In the boolean expressions, the implicit parameter *self* refers to the instance of the TypeName on which the operation is applied. Another predefined parameter, *result*, denotes the return value of the operation if there is one. The dot notation is used to access the attributes of an object or to navigate from that object to the associated objects in a related type. The value of an accessed attribute or role in a postcondition is the value upon completion of the operation. To refer to the value of that property at the start of the operation, one has to postfix the property name with the keyword *@pre*.

As an example consider the diagram of Fig. 2 aimed at representing a set of web portals for selling the products of a company to a group of registered customers, some of them classified as gold customers. The model includes two



context Product **inv** minStock: self.stock $\geq$ 5

context GoldCustomer **inv** salesAmount:
  self.sale $->$select(s | s.paid).cost $->$sum() $\geq$ 100000

context Customer::newCustomer(name:String, p: Portal): Customer
post: result.oclIsNew() and result.name=name and result.portal=p

context Sale::addSaleLine(p: Product, quantity: Integer): SaleLine
pre: p.stock > 0
post: result.oclIsNew() and result.sale=self and result.product=p and
  result.quantity=quantity and p.stock=p.stock@pre-quantity and
  self.amount=self.amount@pre + quantity*p.price

context Portal::removeGoldCategory(c: Customer)
pre: c.oclIsTypeOf(GoldCustomer) and c.sale$->$isEmpty()
post: not c.oclIsTypeOf(GoldCustomer)

Fig. 2. Running example: class diagram, OCL constraints and operations

textual integrity constraints and three operations. The invariant *minStock* ensures that all products have a stock of at least five units, while *salesAmount* imposes that gold customers must have paid a minimum amount of 100000 euros in sales. Regarding the operations, *newCustomer* and *addSaleLine* create a new customer and a new sale line in a sale, respectively. In OCL, the creation of an object is indicated with the operation *oclIsNew*. Operation *addSaleLine* also updates the stock of the product and the total amount of the sale. The operator *@pre* in *p.stock=p.stock@pre - quantity* indicates that the stock of the product has been decreased by *quantity* units with respect to the previous value. *RemoveGoldCustomer* converts a gold customer with no sales into a plain one.

3 List of Correctness Properties

Pre and postconditions of declarative operations must be defined accurately, taking into account the possible interactions with the integrity constraints. For instance, preconditions which are too strong may prohibit the execution of an operation altogether (since none of the valid states of the system can satisfy the precondition). This section presents a list of properties to determine whether pre and postconditions are correctly defined.

In the definition of the correctness properties, we will use the following notation. Given a model M, let S denote a *snapshot* of M, i.e. a possible instantiation of the types defined in M. A snapshot S will be called *legal*, denoted as $\mathsf{Inv}[S]$, if it satisfies all integrity constraints of M, including all textual OCL constraints.

Given a declarative operation *op*, $\mathsf{Pre}_{op}[o, P, S]$ denotes that the precondition of *op* holds when it is invoked over an object o of an snapshot S using the values in P as argument values for the list of parameters of *op*. For the sake of clarity, we will assume that o is passed as an additional parameter in P, e.g. the first one, expressing then the preconditions simply as: $\mathsf{Pre}_{op}[P, S]$. S and P will be referred collectively as the *input* of the operation.

To evaluate the postcondition, we also need to consider the return value and the snapshot after executing the operation (considering new/deleted objects and links, updated attribute values, etc.). The final snapshot and the return value will be referred as the *output* of the operation. Then, $\mathsf{Post}_{op}[P, S \triangleright S', R]$ will denote that the postcondition of operation *op* holds when S is the snapshot before executing the operation, S' is the snapshot after executing it, P is the list of parameters and R is the return value.

According to this notation, the list of properties is defined as follows:

– **Applicability:** An operation *op* is *applicable* if the precondition is satisfiable, i.e. if there is an input where the precondition evaluates to true.

$$\exists S : \exists P : \mathsf{Inv}[S] \ \wedge \ \mathsf{Pre}_{op}[P, S]$$

– **Redundant precondition:** The precondition of an operation *op* is *redundant* if it is true for any legal input.

$$(\exists S : \mathsf{Inv}[S]) \ \wedge \ (\forall S : \forall P : \mathsf{Inv}[S] \ \rightarrow \ \mathsf{Pre}_{op}[P, S])$$

- **Weak executability:** An operation *op* is *weakly executable* if the postcondition is satisfiable, that is, if there is a legal input satisfying the precondition for which we can find a legal output satisfying the postcondition.

$$\exists S, S' : \exists P : \exists R : \mathsf{Inv}[S] \ \wedge \ \mathsf{Inv}[S'] \ \wedge \ \mathsf{Pre}_{op}[P, S] \ \wedge \ \mathsf{Post}_{op}[P, S \triangleright S', R]$$

- **Strong executability:** An operation *op* is *strongly executable* if, for every legal input satisfying the precondition, there is a legal output that satisfies the postcondition.

$$\forall S : \forall P : \exists S' : \exists R : (\mathsf{Inv}[S] \wedge \mathsf{Pre}_{op}[P, S]) \ \rightarrow \ (\mathsf{Inv}[S'] \wedge \mathsf{Post}_{op}[P, S \triangleright S', R])$$

- **Correctness preserving:** An operation *op* is *correctness preserving* if, given a legal input, each possible output satisfying the postcondition is also legal.

$$\forall S, S' : \forall P : \forall R : (\mathsf{Inv}[S] \ \wedge \ \mathsf{Pre}_{op}[P, S]) \ \rightarrow \ (\mathsf{Post}_{op}[P, S \triangleright S', R] \ \rightarrow \ \mathsf{Inv}[S'])$$

- **Immutability:** An operation *op* is *immutable* if, for some input, it is possible to execute the operation without modifying the initial snapshot.

$$\exists S : \exists P : \exists R : \mathsf{Inv}[S] \ \wedge \ \mathsf{Pre}_{op}[P, S] \ \wedge \ \mathsf{Post}_{op}[P, S \triangleright S, R]$$

- **Determinism:** An operation *op* is *non-deterministic* if there is a legal input that can produce two different legal outputs, e.g. different result values or different final snapshots.

$$\begin{aligned}\exists S, S'_1, S'_2 : \exists P : \exists R_1, R_2 : \ & \mathsf{Inv}[S] \ \wedge \ \mathsf{Inv}[S'_1] \ \wedge \ \mathsf{Inv}[S'_2] \ \wedge \ \mathsf{Pre}_{op}[P, S] \ \wedge \\ & \mathsf{Post}_{op}[P, S \triangleright S'_1, R_1] \ \wedge \ \mathsf{Post}_{op}[P, S \triangleright S'_2, R_2] \ \wedge \\ & (\ (S'_1 \neq S'_2) \ \vee \ (R_1 \neq R_2)\ )\end{aligned}$$

Studying these properties in the running example, we have, for instance, that the precondition of *addSaleLine* is redundant since it is subsumed by the integrity constraint *minStock*. Also, *addSaleLine* is weakly executable but not strongly executable: for those states where *p.stock-quantity<5* the final state will violate the invariant *minStock*. The precondition of *removeGoldCategory* is not applicable since constraint *salesAmount* forces all gold customers to be related to at least a sale. Finally, *newCustomer* is strongly executable but not correctness preserving as it might create a gold customer (instead of a plain one) with no sales, violating *salesAmount*.

It is important to remark that, in general, designers define underspecified postconditions [20]. This means that, given an operation contract, there are usually several final states that satisfy its postcondition. Therefore, most operation contracts will be flagged by our analysis as non-deterministic. To improve the accuracy of the results, designers may want to provide postconditions which are precise enough to characterize the exact set of desired final states. For basic postcondition expressions, an educated guess of the designer's intention can be inferred by analyzing the initial ambiguous postcondition [6,8], and thus, it would be possible to automatically generate a set of additional conditions to define more precisely the desired final state. This is left as further work.

4 Verifying Operations with Constraint Programming

This section presents a systematic and automatic procedure to verify correctness properties of operation contracts using the constraint programming paradigm.

Constraint programming [2, 14] is a declarative approach for describing and solving problems. A problem in constraint programming, called *constraint satisfaction problem* (CSP), is defined as a finite set of *variables*, *domains* (one per variable) and *constraints* over the variables. A *solution* to a CSP is an assignment of values to variables that satisfies all constraints, with each value within the domain of the variable. Constraint programming solvers use efficient backtracking-based techniques to automatically explore the search space and find solutions to the CSP. To ensure termination, the search space must be finite, thus, all variable domains must be finite.

The key idea of our approach is to translate the model, together with its integrity constraints, the desired correctness property and the operation to verify, into a CSP such that by checking whether the generated CSP has a solution we can determine if the operation satisfies the property. Both the translation procedure and the search of a solution for the CSP (performed using existing CSP solvers) are completely automatic and, therefore, all the verification process is transparent to the designer.

In short, with our translation procedure, the set of variables in the generated CSP characterize a possible snapshot of the model, i.e. the variable values represent the objects of the snapshot, their attributes values, their relations, etc. Its constraints ensure that the variable values (i.e. the snapshot) are consistent with the implicit structural UML constraints (e.g. all objects in a subtype must be also instance of its supertype), graphical constraints (e.g. multiplicities) and textual OCL constraints. Pre and postconditions of operations and correctness properties are translated as additional constraints.

Given this set of variables, domains and constraints, the final CSP is organized as a sequence of subproblems to be solved by the constraint solver in order to find a solution for the CSP, and thus, prove the desired correctness property. The exact combination of these subproblems in the CSP depends on the chosen property. For properties regarding the operation precondition, the resolution of the CSP first searches for a legal snapshot which satisfies the operation precondition (this, for instance, proves the *applicability* of the operation). If no solution

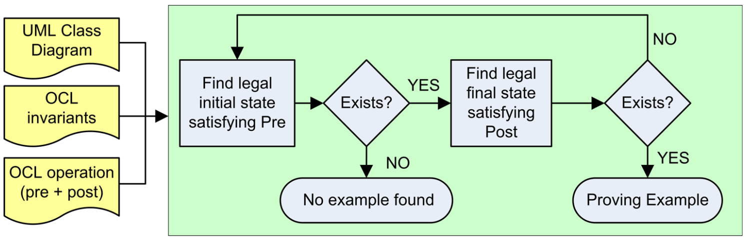

UML Class Diagram
OCL invariants
OCL operation (pre + post)
Find legal initial state satisfying Pre
Exists?
YES
NO
No example found
Find legal final state satisfying Post
Exists?
NO
YES
Proving Example

Fig. 3. Analysis of the weak executabiliy property

is found, the solver concludes that the property is not satisfied. For properties involving postconditions, once we have a legal instance that satisfies the precondition, the solver must search for a second legal snapshot that satisfies the postcondition (see Figure 3). As we will see, for some properties we will search for solutions that falsify the pre/postcondition expressions instead.

The following subsections explain the encoding of the UML class diagram, the OCL constraints and the operations' pre and postconditions in the CSP and how they are combined, depending on the selected correctness property, to generate the final CSP that will be used to prove the property. The first two steps are a short summary of our previous work [9].

Without loss of generality, in our presentation we use the Prolog-based CSP formalism used by the constraint solver ECL^{i}PS^e [2]. Due to space limitations, only some translation excerpts can be shown. The full translation for our running example can be found in [19].

4.1 Translation of the UML Class Diagram

A class diagram consists of a set of classes, associations and generalisation sets. Each element must be translated into a corresponding set of variables, domains and constraints in the CSP. Appropriate domains for each variable can be provided by the designer as part of the translation process or default values can be used.

For each class C, our translation creates a $SizeC$ variable to record the number of instances of C in the snapshot, a list variable $InstancesC$ to hold the C instances, where each element in the list is of type $struct(C) = (oid, f_1, \ldots, f_n)$, where: *oid* represents the explicit object identifier for each object and each field f_i corresponds to an attribute of C.

Similarly, for each association As the translation creates a $SizeAS$ variable to record the number of links (i.e. association instances) in As and a list variable $InstancesAs$ to store the links, where each element is of type $struct(As) = (p_1, \ldots, p_n)$, where $p_1 \ldots p_n$ are the role names of the participant classes. Each concrete participant is identified by its oid.

Generalizations do not imply the definition of new variables but of new constraints among the classes involved in the generalisation set. Additional constraints to control the cardinality constraints or the uniqueness of oid values, among others, are also defined in the CSP.

4.2 Translation of OCL Invariants

Each OCL integrity constraint (*invariants* in the UML terminology) results in a new constraint in the CSP that restricts the possible assignment of values to the CSP variables, i.e. it limits the possible set of legal snapshots of the model.

OCL invariants are boolean OCL expressions defined in the context of a specific type of the model and that must be satisfied by all instances of that type, in other words, the invariant is universally quantified over the objects of the type. Therefore, the translation must ensure that the boolean condition of the invariant (its body) is satisfied by each individual object, i.e. by each possible

```
invariantMinStock(Snapshot) : −
    % Get the list of Objects in Snapshot of type Product
    getObjects(Snapshot, ‘‘Product’’, Objects),
    ( foreach(Object, Objects) do   % Iterate over all objects
      % Evaluate the invariant expression using this object as ‘‘self’’
      evalRootMinStock(Snapshot, [Object], Result),
      % The invariant must evaluate to true
      Result #=1).
```

```
        evalRootMinStock( Snapshot, Vars, Result ) :-
            attribStock( Snapshot, Vars, X ), % X = attrib value
            const5( Snapshot, Vars, Y ),      % Y = constant
            #>=(X, Y, Result).                % Result = X >= Y
        const5( _, _, Result ):- Result #= 5.
```

Fig. 4. Translation of the invariant *minStock* (top) and some subexpressions (bottom)

value of the *self* variable. For instance, the invariant *minStock* (**context** Product **inv** minStock: self.stock $\geq$ 5) would be translated[1] into the rule depicted in Figure 4. This rule fails when the given snapshot contains a product with a too low stock. An auxiliary rule, *evalRootMinStock*, is responsible for checking this condition body on each object. The failure of the rule determines that the snapshot is not legal, and thus, forces the solver to backtrack and try a different combination of variable assignments.

The translation of the body conditions proceeds as follows. The OCL body expression is analyzed using a metamodel-based representation of the expression where each element (operator, variable, constant, method call, . . .) is automatically defined as instance of the appropriate class in the OCL metamodel. Intuitively, an instance of the OCL metamodel for an OCL expression is the equivalent of an annotated syntax tree for the expression. Internal nodes correspond to operators, while the leaves of the tree are constants and variables. The information annotated on each node depends on its type as, for instance, the specific OCL operator, the value of the constant or the identifier of a variable.

The transformation of an OCL expression tree into an ECL^{i}PS^{e} CSP is performed by traversing the tree in postorder and translating each visited node into one Prolog rule with an unique name. For instance, in the invariant *minStock*, *evalRootMinStock* refers to the rule created for the topmost node of the *minStock* invariant body expression. Therefore, the transformation can be fully characterized by describing the Prolog rule that corresponds to each type of node in the OCL metamodel.

[1] To favour the readability of the rules some technical details are omitted.

Prolog rules for OCL expressions follow the pattern:

```
rule-name( Snapshot, Variables, Result ) :- rule-body.
```

where `rule-name` is the unique name of the rule, `Snapshot` and `Variables` are the input arguments and `Result` stores the output of the expression. Intuitively, `Snapshot` is the snapshot of the model where the expression is evaluated. `Variables` is the list of variables visible in the scope of the expression, e.g. the *self* variable and variables introduced by previous quantifiers due to iterator expressions like *forAll*. In the `rule-body` we specify the sequence of predicates that describe the relationship between the inputs and the output. A typical body evaluates the subexpressions (using their Prolog rule) and computes the output from those intermediate results.

As an example, let us consider the body of the invariant *minStock* (self.stock $\geq$ 5), which contains four subexpressions: a variable (self), an attribute access (stock), a constant (5) and a relational operator ($\geq$). The rules for the last two expressions are depicted in Figure 4 (see [9, 19] for more examples). For more complex OCL operators and iterator expressions we have already implemented a parametrized library of Prolog rules (available in [19]) that maps the semantics of each predefined OCL construct.

4.3 Translation of OCL Operation Contracts

Operations introduce new challenges in this translation: the list of parameters of the operation, the result value, and the complexity of studying two snapshots at once when analyzing postconditions.

Translation of preconditions. The boolean OCL expression of a precondition is basically translated following the same procedure explained above for the translation of invariant bodies. However, there are two differences regarding how and when the precondition expression is evaluated: the *parameters* and the *quantification*.

In the analysis of a precondition, it is necessary to consider the possible value of the operation parameters. For parameters of a basic type (integer, float, boolean, string) designers must define their possible finite domain, for instance defining a lower and upper bound. Parameters whose type is one of the classes of the model (as the *self* parameter) can only refer to an object existing in the snapshot, so their value is already constrained by the valid instances of the snapshot where the operation is invoked. When evaluating a precondition, parameters become additional variables of the CSP, and their values are discovered by the solver as a part of the search for a solution to the CSP. For instance, when checking the applicability of an operation, the solver will automatically try several possible combinations of parameter values until it finds a combination (if any) that satisfies the Prolog rule generated for the precondition.

Contrary to invariants, properties on preconditions only require to find a single combination of a valid state and a possible assignment for the operation parameters that satisfy the precondition. Therefore, preconditions will be translated into a rule which simply evaluates the precondition body, invoking the rule for the topmost operator. To ensure that the rules for the precondition body have access to all parameter values during the rule evaluation, the list of visible variables for these rules (second argument of the Prolog rule) is initialized with the list of parameter values. In this way, accessing a parameter within the expression is equivalent to accessing any other variable: the rule only needs to be aware of the position of each parameter in the variables list. As an example, the precondition rule for *addSaleLine* will be defined as follows:

```
preconditionAddSaleLine(Snapshot, Parameters, Result) :-
    % Result = truth value of evaluating the precondition
    evalRootExpr(Snapshot, Parameters, Result).
```

where *evalRootExpr* represents the rule for the root node of the precondition expression. The output *Result* value, reporting whether the given input (i.e. the *self* object plus the other parameters) satisfies the precondition, will be used later on to determine the satisfaction of correctness properties for the operation.

Translation of postconditions. Two new factors in the translation of postconditions are the return value and the relationship between the two snapshots representing the initial and final states.

In our translation, the return value will simply become another variable in the list of visible variables, just like *self* or the other parameters in the precondition.

Relationships between the initial and the final state are expressed by means of the *oclIsNew* and, specially, the *@pre* OCL operators. *OclIsNew* highlights that an object should exist in the final state but not in the initial one; and *@pre* is used to retrieve the value of a subexpression in the initial state. Thus, the Prolog implementation of these two operators needs to receive an additional argument: the snapshot for the initial state. To avoid changing the general rule pattern due to this extra argument, this initial state is stored in the global variable `initialstate`. This variable will be conveniently accessed within the subrules for these two operators. Translation of all other OCL operators in the postcondition expression is not changed from previous translations steps. They are just evaluated on the particular snapshot given as argument to their Prolog rule, it does not matter if it represents the initial or the final state.

To sum up, the definition of the rule for the postcondition of the operation *addSaleLine* is shown in Figure 5. The *initialstate* variable will then be used in the rules evaluating *oclIsNew* and *@pre* nodes appearing in postcondition expressions. We provide the rule for *oclIsNew* as an example in Figure 6. It determines if the object with the *Oid* value given as an argument is an object that did not exist before executing the operation.

```
:- local reference(initialstate).
postconditionAddSaleLine(InitialState, FinalState,
                         Parameters, RetValue, Result) :-
    % Add the return value and parameters to the list of visible vars
    append([RetValue], Parameters, Variables),
    % Store the initial state, needed in oclIsNew and @pre nodes
    setval(initialstate, InitialState),
    % Result = truth value of evaluating the postcondition
    evalRootExpr(FinalState, Variables, Result).
```

Fig. 5. Translation of the OCL postcondition of operation *addSaleLine*

```
ocl_isNew(FinalState, Oid, TypeName, Result) :-
    % Recover the initial state from the global variable
    getval(initialstate, InitState),
    % Get the list of objects before and after the operation
    getObjects(InitState, TypeName, ObjectsBefore),
    getObjects(FinalState, TypeName, ObjectsAfter),
    % Check if Oid exists before/after the operation
    existsObjectWithOid(ObjectsBefore, Oid, ExistsBefore),
    existsObjectWithOid(ObjectsAfter, Oid, ExistsAfter),
    % Result = ExistsAfter and not ExistsBefore
    and( ExistsAfter, neg ExistsBefore, Result).
```

Fig. 6. Translation of the OCL operator *oclIsNew*

4.4 Translation of Correctness Properties

As a last step, each correctness property (or its negation) is translated as a new CSP constraint restricting the *result* values returned by the pre and postcondition rules such that finding a solution to the CSP with this new constraint suffices to prove the property.

Whether to use the property or its negation depends on the quantification used in the property formalization, *existential* or *universal* (see Section 3). Existentially quantified properties can be *proved* by finding an *example*, i.e. a case where the property is satisfied. For example, applicability can be proved by finding a legal input that satisfies the precondition. Universally quantified properties can be *disproved* by finding a *counterexample*. For instance, redundancy can be disproved by finding a legal snapshot that does not satisfy the precondition Similarly, the lack of (counter)examples can be used to (dis)prove the property.

```
weakExecutabilityAddSaleLine(Example) :-
    Example = [InitState, FinalState, Parameters, RetValue],
    findInitialState(InitState, Parameters),
    findFinalState(InitState, FinalState, Parameters, RetValue).
findInitialState(InitState, Parameters) :-
    % Definition of variables, domains, graphical integrity constraints
    % Textual integrity constraints
    invariantMinStock(InitState), invariantSalesAmount(InitState),
    % Precondition
    preconditionAddSaleLine(InitState, Parameters, ResultOfPre),
    ResultOfPre #= 1, % Weak executability
    % Now find a solution satisfying all these constraints
    labeling([InitState, Parameters]).
findFinalState(InitState, FinalState, Parameters, RetValue) :-
    % Definition of variables, domains, graphical integrity constraints
    % Textual integrity constraints
    invariantMinStock(FinalState), invariantSalesAmount(FinalState),
    % Postcondition
    postconditionAddSaleLine(InitState, FinalState, Parameters,
                             RetValue, ResultOfPost),
    ResultOfPost #= 1, % Weak executability
    % Now find a solution satisfying all these constraints
    labeling([FinalState, RetValue]).
```

Fig. 7. CSP generated for checking weak satisfiability of *addSaleLine*. The *labeling* operator is a possible backtracking implementation offered by the constraint solver that attempts to assign values to the given list of input variables. If the assignment does not satisfy all the stated CSP constraints preceding the labeling, a new assignment is tried until the solver finds a solution or determines that no solution exists.

The selected property also influences how the final CSP is organized as a combination of the rule excerpts generated during the previuos translation steps. For properties on preconditions, postcondition rules are not included. For properties on postconditions, the CSP is split up into two subproblems (see Figure 3). The first one (*findInitialState*) tries to find a legal snapshot that satisfies the precondition rule. This initial snapshot is then given as an argument to the second subproblem (*findFinalState*), in charge of finding a second legal snapshot satisfying (or not) the postcondition to prove the property. As an example, Figure 7 sketches the final CSP to determine whether *addSaleLine* is weakly executable. Other properties imply adding new constraints/subproblems to the CSP. For instance, immutability requires a new constraint imposing the equality between the initial and final states.

5 Tool Support

The verification method presented in this paper is being implemented as an extension of our UMLtoCSP tool [19]. Given an UML/OCL model and a correctness property P, the tool determines whether the model satisfies P and shows graphically example snapshots that prove/disprove it. For instance, Fig. 8 shows the counterexample provided by UMLtoCSP when analyzing whether *addSaleLine* is correctness preserving: there is a legal input (the snapshot on the left satisfies the invariants and precondition) with an illegal output (the snapshot on the right satisfies the postcondition but not the invariant *minStock* as there are only 4 items of *Product*1). The translation from the model to the CSP, the search of the counterexample snapshots and the graphical depiction are performed automatically by UMLtoCSP. See [19] for more details and examples.

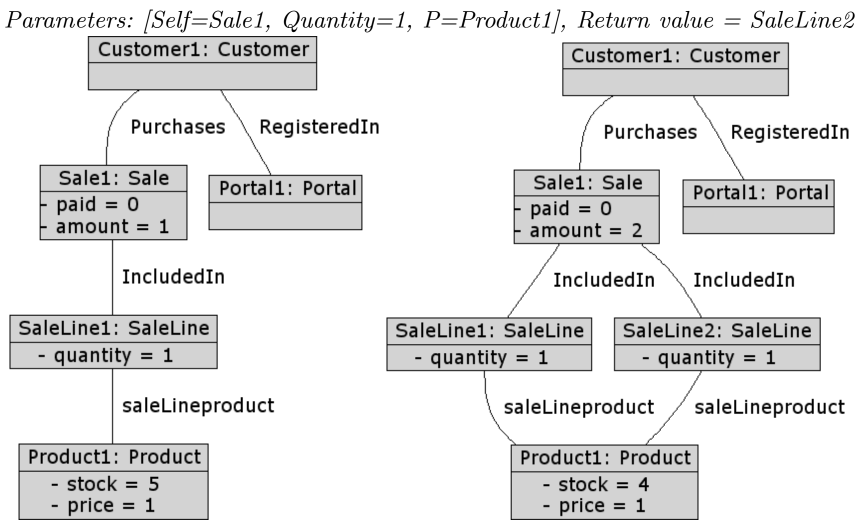


Fig. 8. Counterexample proving that *addSaleLine* is not correctness preserving: initial state (left), final state (right), parameter values and return value (top). The final state violate the invariant *minStock*.

6 Related Work

Typically, approaches devoted to the verification of UML/OCL models (as [1,4, 7,10,13,16,17] or our own approach among others) transform the diagram into a formalism where efficient solvers or theorem provers are available. However, there are complexity and decidability issues to be considered. Reasoning on UML class diagrams is EXPTIME-complete [3], and, when general OCL constraints and/or operations are allowed, it becomes undecidable.

By choosing a particular formalism, each method commits to a different trade-off (expressivity vs termination vs automation vs completeness) for the verification process. In what follows we compare the features of methods supporting the verification of declarative operations, a small subset of the ones listed above, with this paper.

HOL-OCL [7] embeds OCL into the higher-order logic (HOL) instance of the interactive theorem prover Isabelle. It supports the full OCL expressivity but it requires user-interaction to complete proofs, and thus, it is not automatic.

The UMLtoAlloy tool [1] proposes an automatic translation of UML/OCL to Alloy [12]. Alloy is a mature tool for the automated analysis of software specifications that works by transforming the entire problem, including operation specifications, into an instance of SAT (satisfiability of a boolean formula in conjunctive normal form). However, the translation in [1] is only partial and Alloy itself presents some limitations, such as the need explicitly identify which model elements are modified by an operation or limited support for arithmetic operations. Thus, the usefulness of Alloy for verifying high-level UML/OCL specifications is somewhat limited.

Recent results [11] have extended the description logics formalism (in short, a decidable subset of first-order logic) to define and reason on operation contracts. However, these approaches need to restrict the constructs that may appear in the model to keep the reasoning decidable. Thus, most OCL operations cannot be translated into this formalism.

Previous approaches based on constraint programming like [10,13] did not admit any kind of OCL expressions. Our previous work in [9,19] was limited to OCL invariants and did not support the analysis of declarative operations.

In contrast, the new approach presented in this paper is fully automatic, expressive and decidable. We believe that these three characteristics are key features in order to extend the adoption of formal methods among the modeling community. As a trade-off, our method is not complete: results are only conclusive if a solution to the CSP (the example/counterexample) is found. However, the absence of solutions within a finite search space cannot be used as a proof: a solution may still exist outside that search space.

Although this may limit the applicability of our method, we believe an efficient decidable procedure provides more useful information than a semidecidable procedure, even if the answer is not conclusive. Moreover, when checking the correctness of a model, most errors can be found even if we bound the search space of the verification process. This "small scope" hypothesis, i.e. that it is possible to prove interesting properties about models by focusing only on small instantiations, is shared by other bounded methods [12]. Moreover, if desired, it is still possible to use our method on infinite domains [2] resulting in a complete but semidecidable method (for properties that can be satisfied by *finite* instances).

Our approach can be complemented with other verification approaches addressing other kinds of behavioural UML diagrams (as state machines [18]) in order to provide more global results.

7 Conclusions and Further Work

We have presented a new automatic method for the formal verification of declarative operations in UML/OCL models. We believe our approach can be used to leverage current UML/OCL verification approaches, more focused on the verification of the static parts of the model.

Regarding efficiency, the search space for examples/counterexamples depends on the size of the model, so scalability quickly becomes an issue even when using sofisticated constraint solvers. As a further work, we plan to improve the search process in several ways. First, we would like to refine our translation process by considering implicit semantics in the initial contract specification (as the *nothing else changes assumption*). Also, we plan to work on the automatic inference of variable domains, discovered by a static analysis of the OCL constraints to tune the solving process. Furthermore, we are considering the abstraction of information from the model which is not relevant to the operation being verified and the relevant subset of integrity constraints.

We also plan to explore the verification of dynamic aspects of the model when specified in combination with other constructs or UML diagrams like sequence diagrams or state machines and the benefits of porting these techniques to other design by contract languages as JML, Eiffel or Spec#.

References

1. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: UML2Alloy: A challenging model transformation. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 436–450. Springer, Heidelberg (2007)
2. Apt, K.R., Wallace, M.G.: Constraint Logic Programming using ECL^{i}PS^e. Cambridge University Press, Cambridge (2007)
3. Artale, A., Calvanese, D., Kontchakov, R., Ryzhikov, V., Zakharyaschev, M.: Reasoning over extended ER models. In: Parent, C., Schewe, K.-D., Storey, V.C., Thalheim, B. (eds.) ER 2007. LNCS, vol. 4801, pp. 277–292. Springer, Heidelberg (2007)
4. Berardi, D., Calvanese, D., Giacomo, G.D.: Reasoning on UML class diagrams. Artificial Intelligence 168, 70–118 (2005)
5. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley, Reading (1998)
6. Borgida, A., Mylopoulos, J., Reiter, R.: On the frame problem in procedure specifications. IEEE Trans. Software Eng. 21(10), 785–798 (1995)
7. Brucker, A.D., Wolff, B.: The HOL-OCL book. Technical Report 525, ETH Zurich (2006)
8. Cabot, J.: From declarative to imperative UML/OCL operation specifications. In: Parent, C., Schewe, K.-D., Storey, V.C., Thalheim, B. (eds.) ER 2007. LNCS, vol. 4801, pp. 198–213. Springer, Heidelberg (2007)
9. Cabot, J., Clarisó, R., Riera, D.: Verification of UML/OCL class diagrams using constraint programming. In: IEEE International Conference on Software Testing Verification and Validation Workshop, ICSTW 2008, pp. 73–80 (2008)

10. Cadoli, M., Calvanese, D., Giacomo, G.D., Mancini, T.: Finite satisfiability of UML class diagrams by Constraint Programming. In: DL 2004. CEUR Workshop Proceedings, vol. 104, CEUR-WS.org (2004)
11. Drescher, C., Thielscher, M.: Integrating action calculi and description logics. In: Hertzberg, J., Beetz, M., Englert, R. (eds.) KI 2007. LNCS, vol. 4667, pp. 68–83. Springer, Heidelberg (2007)
12. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Transactions on Software Engineering and Methodology 11(2), 256–290 (2002)
13. Malgouyres, H., Motet, G.: A UML model consistency verification approach based on meta-modeling formalization. In: Biham, E., Youssef, A.M. (eds.) SAC 2006. LNCS, vol. 4356, pp. 1804–1809. Springer, Heidelberg (2007)
14. Marriott, K., Stuckey, P.J.: Programming with Constraints: An Introduction. MIT Press, Cambridge (1998)
15. Object Management Group. UML 2.0 OCL Specification (2003)
16. Queralt, A., Teniente, E.: Reasoning on UML class diagrams with OCL constraints. In: Embley, D.W., Olivé, A., Ram, S. (eds.) ER 2006. LNCS, vol. 4215, pp. 497–512. Springer, Heidelberg (2006)
17. Straeten, R.V.D., Mens, T., Simmonds, J., Jonckers, V.: Using description logic to maintain consistency between UML models. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 326–340. Springer, Heidelberg (2003)
18. Turner, E., Treharne, H., Schneider, S., Evans, N.: Automatic generation of CSP B skeletons from xuml models. In: Fitzgerald, J.S., Haxthausen, A.E., Yenigun, H. (eds.) ICTAC 2008. LNCS, vol. 5160, pp. 364–379. Springer, Heidelberg (2008)
19. UMLtoCSP. A tool for the formal verification of UML/OCL models based on Constraint Programming, `http://gres.uoc.edu/UMLtoCSP`
20. Wieringa, R.: A survey of structured and object-oriented software specification methods and techniques. ACM Comput. Surv. 30(4), 459–527 (1998)

Property Specifications for Workflow Modelling

Peter Y.H. Wong and Jeremy Gibbons

Computing Laboratory, University of Oxford, United Kingdom
{peter.wong,jeremy.gibbons}@comlab.ox.ac.uk

Abstract. Previously we provided two formal behavioural semantics for Business Process Modelling Notation (BPMN) in the process algebra CSP. By exploiting CSP's refinement orderings, developers may formally compare their BPMN models. However, BPMN is not a specification language, and it is difficult and sometimes impossible to use it to construct behavioural properties against which BPMN models may be verified. This paper considers a pattern-based approach to expressing behavioural properties. We describe a property specification language *PL* for capturing a generalisation of Dwyer et al.'s Property Specification Patterns, and present a translation from *PL* into a bounded, positive fragment of linear temporal logic, which can then be automatically translated into CSP for simple refinement checking. We demonstrate its application via a simple example.

1 Introduction

Formal developments in workflow languages allow developers to describe their workflow systems precisely, and permit the application of model checking to automatically verify models of their systems against formal specifications. One of these workflow languages is the Business Process Modelling Notation (BPMN) [6], for which we previously provided two formal semantic models [8,9] in the process algebra CSP [7]. Both models leverage the refinement orderings that underlie CSP's denotational semantics, allowing BPMN to be used for specification as well as modelling of workflow processes. However, due to the fact that the expressiveness of BPMN is strictly less than that of CSP, some behavioural properties, against which developers might be interested to verify their workflow processes, might not be easy or even possible at all to capture in BPMN.

As a running example for this paper, consider the BPMN diagram describing a travel agent shown in Figure 1. The main purpose of the travel agent is to mediate interactions between the traveller who wants to buy airline tickets and the airline who supplies them. Specifically, once the travel agent receives an initial order from the traveller (*Receive_Order*), he needs to verify with the airline if the seats are available for the desired trip (*Check_Seats*). In order to cater for the possibility of the traveller making changes to her itinerary, for every change of her itinerary (*Change_Itin_TA*), the travel agent verifies with the airline the availability of the seats (*Check_Seats_2*). Once the traveller has agreed upon a particular itinerary (*Receive_Reservation*), the travel agent reserves the seats for the traveller (*Reserve_Seats*). During the reservation period, modelled by the *Reservation*

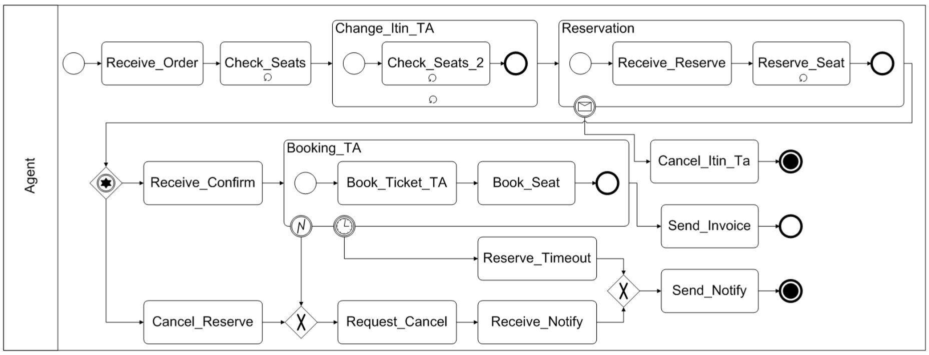


Fig. 1. Travel Agent

subprocess state, the traveller may cancel her itinerary, thereby "unreserving" the seats; this is modelled as a message exception flow (*Imessage*) of the *Reservation* subprocess. Once the reservation has been completed, the travel agent may receive a confirmation notice from the traveller (*Receive_Confirm*), in which case he receives the credit card information from the the traveller (*Book_Ticket_TA*) and proceeds with the booking (*Book_Seat*). The travel agent may also receive cancellation of the reservation (*Cancel_Reserve*), in which case he will request a cancellation from the airline (*Request_Cancel*), wait for a notification confirming the cancellation from the airline (*Receive_Notify*), and send it to the traveller (*Send_Notify*). During the booking phase, either an error (e.g. incorrect card information) or a time out (*Reserve_Timeout*) may occur; in both cases, corresponding notification confirming the cancellation will be sent to the traveller. Otherwise, a corresponding invoice on the booking will be sent to the traveller for billing (*Send_Invoice*).

One of the properties this travel agent description must satisfy is that the agent must not allow any kind of cancellation after the traveller has booked her tickets, if invoice is to be sent to the traveller. Assuming process *Agent* models the semantics of the travel agent diagram, one might attempt to draw a BPMN diagram like the one shown in Figure 2(a) to express the negation of the property, and prove the satisfiability of *Agent* by showing this diagram does not failures-refine the process $Agent \setminus N$ where N is the set of CSP events that are not associated with tasks *Book_Seat*, *Request_Cancel*, *Request_Timeout* and *Send_Invoice*. However, while this behavioural property should also permit other behaviours such as task *Request_Cancel* being performed before task *Book_Seat*, it could be difficult to specify all these behaviours in the same BPMN diagram. Since BPMN is a modelling notation for describing the *performance* of behaviour, in general it is difficult to use it to specify liveness properties about the *refusal* of some behaviour within a context while asserting the *availability* of it outside the context. We therefore need a different approach which will allow domain specialists to express property specifications for verification of workflow processes.

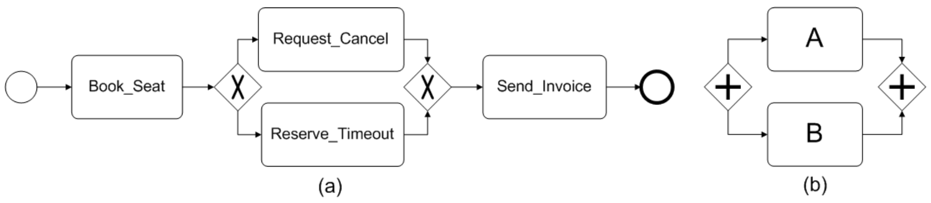


Fig. 2. (a) A BPMN diagram capturing requirement and (b) Parallel execution

1.1 Property Specification Patterns

This paper proposes the application of Dwyer et al's *Property Specification Patterns* [1] to assist domain specialists to specify behavioural properties for BPMN processes[1]. Specification patterns are generalised specifications of properties for finite-state verification. They are intended to describe the essential structure of commonly occurring requirements on the permissible patterns of behaviours in a finite state model of a system. There exist two major groups – *order* and *occurrence*. Each pattern has a scope, the context in which the property must hold. For example, the property "task A cannot happen after task B and before task C" will fall into the *absence* pattern, which states that a given state/event does not occur within a scope. In this case, the property may be expressed as the absence of task A in the scope *after task B until task C*. The different types of scope are *Global*, *Before Q*, *After Q*, *Between Q and R* and *After Q until R*, where Q and R are states.

Currently, property patterns have been expressed in a range of formalisms such as linear temporal logic (*LTL*) [4] and computation tree logic; however, behavioural verifications of CSP processes are carried out by proving a refinement between the specification and the implementation processes. This means CSP is also a *specification language*, and to the best of our knowledge there is currently no formalisation of property patterns in CSP.

1.2 Nondeterministic Interleaving

While the property patterns cover a comprehensive set of behavioural requirements, it is possible to generalise patterns in a process-algebraic setting by considering *patterns of behaviour* rather than an individual state or event within a scope. For example, we may like to express the property "the parallel execution of task A and either task D or task E cannot happen after task B and before task C". Here the pattern of behaviours is "the parallel execution of task A and either task D or task E". While CSP is equipped with nondeterministic choice as one of its standard operators, there is no nondeterministic version of parallel composition; this means that while assertion (1) holds under failures refinement, assertion (2) does not.

[1] We assume readers have basic knowledge of CSP and that they are not required to have knowledge of BPMN.

$$a \rightarrow \mathit{Skip} \sqcap b \rightarrow \mathit{Skip} \sqsubseteq_{\mathcal{F}} a \rightarrow \mathit{Skip} \tag{1}$$

$$a \rightarrow \mathit{Skip} \mathbin{|||} b \rightarrow \mathit{Skip} \sqsubseteq_{\mathcal{F}} a \rightarrow b \rightarrow \mathit{Skip} \tag{2}$$

This is because the parallel operators in CSP may be defined using the deterministic choice $\Box$ operator; here we show the interleaving of two processes and its equivalent sequential counterpart.

$$a \rightarrow \mathit{Skip} \mathbin{|||} b \rightarrow \mathit{Skip} \equiv a \rightarrow b \rightarrow \mathit{Skip} \Box b \rightarrow a \rightarrow \mathit{Skip}$$

A nondeterministic version of the parallel operators, particularly interleaving, may be very useful for specifying behavioural properties for workflow processes. For example, Figure 2(b) shows part of a BPMN diagram executing tasks A and B in parallel. With our timed semantics for BPMN [9] it is possible to specify timing constraints for these tasks, and the diagram may then be interpreted over the timed model. It is easy to see the possibility of one asserting a behavioural property about tasks A and B within a wider scope without considering the ordering of their execution due to their timing constraints.

1.3 Our Approach

Our objective is to provide a CSP formalisation of the set of *generalised* property specification patterns, in which we consider admissible sequences of patterns of behaviours, rather than individual events, within a scope. The construction of the CSP model for each of the patterns proceeds in two stages:

– we first define a small property specification language *PL*, based on the generalised patterns, for describing behavioural properties, and then provide a function that returns a linear temporal logic (*LTL*) expression that specifies the behaviour properties;
– we then translate the given *LTL* expression into its corresponding CSP process based on Lowe's interpretation of *LTL* [3]; using this, one may check whether a workflow system behaves according to a property specification.

Specifically we provide a function which translates each of the property patterns into the *bounded, positive fragment* of *LTL* [3], denoted by *BTL*, defined by the following grammar.

$$\begin{array}{ll} \phi \in \mathit{BTL} ::= \phi \wedge \phi \mid \phi \vee \phi \mid \bigcirc\phi \mid \Box\phi \mid \phi \mathcal{R} \phi \mid a \mid \neg a \mid & \quad \textbf{where}\; a \in \Sigma \\ \qquad \texttt{available}\; a \mid \texttt{true} \mid \texttt{false} \mid \texttt{live} \mid \texttt{deadlocked} & \end{array}$$

where operators $\neg$, $\wedge$ and $\vee$ are standard logical operators, and $\bigcirc$, $\Box$ and $\mathcal{R}$ are standard temporal operators for *next*, *always* and *release*. This fragment also extends the original logic with atomic formulae for specifying *availability* of events as well as their performance. Here we describe briefly their intended meaning:

– a – the event a is available to be performed initially, and no other events may be performed;

- `available` a – the event a must not be refused initially, and other events may be performed;
- `live` and `deadlock` – the system is live (equivalent to $\bigvee_{a \in \Sigma} a$) or deadlocked (equivalent to $\bigwedge_{a \in \Sigma} \neg a$), respectively;
- `true` and `false` – logical formulae with their normal meanings.

Usually when checking whether a (workflow) system, modelled as a CSP process, satisfies a certain behavioural property, which is also modelled as a CSP process, one would check to see the former refines the latter under the stable failures semantics [7], since this model captures both safety and liveness properties. However, Lowe [3] has shown that the stable failures model is not sufficient to capture temporal logic specifications, and that a finer model known as the refusal traces model ($\mathcal{RT}$) [5] is required. Furthermore, Lowe has also shown that it is impossible to capture the *eventually* ($\Diamond$) and *until* ($\mathcal{U}$) temporal operators as well as the negation operator ($\neg$) in general. This is because the eventual operator deals with *infinite traces*, which are not suitable in general in finite-state checking, and since $\Diamond\, \phi = \mathtt{true}\, \mathcal{U}\, \phi$, it is also not possible, in general, to capture the *until* operator. Also $\Diamond\, \phi = \neg(\Box\neg\phi)$ and it is possible to capture the *always* operator, therefore it is not possible, in general to capture negation, unless only over atomic formulae as given by the grammar above. Our function reflects this by translating a given generalised pattern into a corresponding expression *BTL*. We say a system modelled by the CSP process P satisfies a behavioural property, written as $P \models \psi$ where ψ is the temporal logic expression, if and only if $\mathit{Spec}(\psi) \sqsubseteq_{\mathcal{RT}} P$ where $\mathit{Spec}(\psi)$ is the CSP specification for ψ.

1.4 Assumptions and Structure of the Paper

In the rest of this paper we assume the behaviour of the system we are interested in is modelled by some non-divergent process P. We assume the alphabet of the *specification process* of the property, that is the set of all possible events the process may perform, only falls under the context of the property. This is possible because in CSP, one may always construct some *partial specification* X and prove some system Y satisfies it by checking the refinement assertion $X \sqsubseteq Y \setminus (\alpha Y \setminus \alpha X)$ where αP is the alphabet of P, assuming $\alpha X \subseteq \alpha Y$.

The structure of the remainder of this paper is as follows. Section 2 gives a brief overview of the refusal traces model. In Section 3 we introduce *SPL*, a sub-language of our property specification language, for specifying nondeterministic patterns of behaviours; we define function *pattern*, which takes a nondeterministic system specified in *SPL* and returns its corresponding temporal logic expression in *BTL*. We provide justification for the translation over the refusal traces model. In Section 4 we present the complete language *PL* for specifying behavioural properties based on generalised property patterns. We then define a function *makeTL*, which takes a property specification in *PL* and returns its corresponding temporal logic expression in *BTL*, and finally we revisit the travel agent running example and demonstrate how to specify the behavioural property in *PL*.

2 Refusal Traces Model

CSP [7] is equipped with three standard behavioural models: traces, stable failures and failures-divergences, in order of increasing precision. However, Lowe [3] has demonstrated that these models are inadequate for capturing temporal logic of the form described in previous section. The solution is to use the refusal traces model ($\mathcal{RT}$) [5].

In the refusal traces model, each CSP process may be denoted as a set of refusal traces; each refusal trace is an alternating sequence of refusal information and events. More precisely, a refusal trace takes the form $\langle X_1, a_1, X_2, a_2, .., X_n, a_n, \Sigma \rangle$, where each X_i is a refusal set, and each a_i is an event. This test represents that the process can refuse X_1, perform a_1, refuse X_2, perform a_2, etc. In this particular example the refusal trace finishes by refusing Σ (the set of all possible events), i.e. deadlocking.

Here we write $\mathcal{RT}[\![P]\!]$ for the refusal traces of CSP process P. We now present the refusal traces semantics for some of the CSP operators,

$$\begin{aligned}
\mathcal{RT}[\![\mathit{Stop}]\!] &= \{\, \langle\rangle, \langle \Sigma \rangle \,\} \\
\mathcal{RT}[\![a \to P]\!] &= \{\, \langle\rangle \,\} \cup \{\, \langle X, a \rangle ^\frown tr \mid a \notin X \wedge tr \in \mathcal{RT}[\![P]\!] \,\} \\
\mathcal{RT}[\![P \sqcap Q]\!] &= \mathcal{RT}[\![P]\!] \cup \mathcal{RT}[\![Q]\!] \\
\mathcal{RT}[\![P \,\Box\, Q]\!] &= \{\, \langle\rangle \} \cup (\textbf{if } \langle \Sigma \rangle \in \mathcal{RT}[\![P]\!] \cap \mathcal{RT}[\![Q]\!] \textbf{ then } \{\, \langle \Sigma \rangle \,\} \textbf{ else } \emptyset) \cup \\
&\qquad \{\, \langle X, a \rangle ^\frown tr \mid \langle X, a \rangle ^\frown tr \in \mathcal{RT}[\![P]\!] \wedge Q \,\textbf{ref}\, X \,\vee \\
&\qquad\qquad\qquad \langle X, a \rangle ^\frown tr \in \mathcal{RT}[\![Q]\!] \wedge P \,\textbf{ref}\, X \,\}
\end{aligned}$$

where $Q\,\textbf{ref}\,X$ means that Q can refuse X initially.

Refinement in the refusal traces model is then defined as follows:

$$\mathit{Spec} \sqsubseteq_{\mathcal{RT}} P \Leftrightarrow \mathcal{RT}[\![\mathit{Spec}]\!] \supseteq \mathcal{RT}[\![P]\!]$$

Currently the CSP model checker, FDR [2], is being extended to include the checking of refinement in this model.

3 Patterns of Behaviour

Here we present a sub-language of our property specification language *PL*, denoted as *SPL*, for assisting developers to construct BPMN-based patterns of behaviour:

$$\begin{aligned}
&P \in \mathit{SPL} ::= P \sqcap P \mid P \sqcap\!\!\sqcap P \mid a \to P \mid \mathit{End} \quad \textbf{where}\; a \in \mathit{Atom} \\
&\mathit{Atom} \qquad ::= t \mid \texttt{available}\; t \mid \texttt{live} \quad \textbf{where}\; t \in \mathit{Task}
\end{aligned}$$

where the basic type *Task* represents the set of names that identify task states in a BPMN diagram, and the type *Atom* describes the *performance* or the *availability* of some task t. The behaviour $t \to P$ hence enacts task t and then behaves like P. The atomic term `live` describes the *performance* of any task state of the BPMN diagram in question. An user interface for this language could be implemented to assist BPMN developers to construct specifications.

The language is equipped with operators focusing on specifying nondeterministic concurrent systems that are suitable as process-based specifications. Specifically it contains a subset of standard CSP operators, that is nondeterministic choice ($\sqcap$) and prefix ($\rightarrow$), as well as a new *nondeterministic interleaving* operator ($\sqcap\!\!\sqcap$). Informally the process $P \sqcap\!\!\sqcap Q$ communicates events from both P and Q, but unlike CSP's interleaving, our operator chooses them nondeterministically. Here we present the step law governing the operator in the form of CSP's algebraic laws [7]: if $P = p \rightarrow P'$ and $Q = q \rightarrow Q'$ then

$$P \sqcap\!\!\sqcap Q = (p \rightarrow (P' \sqcap\!\!\sqcap Q)) \sqcap (q \rightarrow (P \sqcap\!\!\sqcap Q')) \qquad [\sqcap\!\!\sqcap\text{-step}]$$

and we present the laws of this operator over *end*:

$$\mathit{End} \sqcap\!\!\sqcap Q = Q \qquad [\sqcap\!\!\sqcap\text{-End}]$$

The operator $\sqcap\!\!\sqcap$ is both commutative and associative and is defined in terms of nondeterministic choice $\sqcap$ and prefix $\rightarrow$. This operator allows developers to construct patterns of behaviour representing parallel executions of task states without needing to know more refined detail such as timing information which may restrict possible orders of enactments of states.

Now we present the function *pattern*, which takes a pattern of behaviour described in *SPL* and returns the corresponding formula in BTL^*. Here BTL^* denotes *BTL* augmented with the atomic formula $*$, which has the empty set of refusal traces. We write $event(t)$ to denote an event associated with task t. For all $a \in Atom$, $t \in Task$ and $P, Q \in SPL$,

$$\begin{aligned}
&pattern(End) = * \\
&pattern(a \rightarrow P) = atom(a) \land \bigcirc(pattern\ P) \\
&pattern(P \sqcap Q) = pattern(P) \lor pattern(Q) \\
&pattern(P \sqcap\!\!\sqcap Q) = pattern(npar(P, Q))
\end{aligned}$$

where *npar* will be defined shortly and the function *atom* is defined as follows:

$$\begin{aligned}
&atom(\texttt{available}\ t) = \texttt{available}\,(event(t)) \\
&atom(\texttt{live}) = \texttt{live} \\
&atom(t) = event(t)
\end{aligned}$$

Due to this translation *End* has an empty semantics.

To convert formulae in BTL^* back to *BTL*, we simply remove $*$ according to the following equivalences: $\phi \lor * \equiv \phi$, $* \land \phi \equiv \phi$ and $\phi \land \bigcirc * \equiv \phi$; note both the conjunctive and disjunctive operators are commutative.

We map each of the operators other than $\sqcap\!\!\sqcap$ directly into their corresponding temporal logic expression. Here we show that the semantics of the prefix operator $\rightarrow$ is preserved by the translation. First we give the semantic definition of $\rightarrow$ over *SPL* in the refusal traces model $\mathcal{RT}$ where *RT* denotes all (finite) refusal traces. For all $a \in \Sigma$, $X \in \mathbb{P}\,\Sigma$ and $tr \in RT$:

$$\begin{array}{l}
\mathcal{RT}_{SPL}[\![*]\!] = \emptyset \\
\mathcal{RT}_{SPL}[\![t \rightarrow P]\!] = \\
\qquad \{ \langle\rangle \} \cup \{ \langle X, a\rangle \frown tr \mid a = event(t) \wedge a \notin X \wedge tr \in \mathcal{RT}_{SPL}[\![P]\!] \}
\end{array}$$

We write $\langle\rangle$ for the empty sequence, $\langle a, b\rangle$ for a sequence of a followed by b and $s \frown t$ for the concatenation of the sequences s and t. Similarly we present Lowe's semantic definition [3] for the operators $\bigcirc$, $\wedge$ over BTL and the atomic formula a in $\mathcal{RT}$, where IRT denotes the set of all infinite refusal traces. For all $a \in \Sigma$, $X \in \mathbb{P}\,\Sigma$ and $tr \in RT \cup IRT$:

$$\begin{array}{l}
\mathcal{RT}_{BTL}[\![a]\!] = \{ \langle\rangle \} \cup \{ \langle X, a\rangle \frown tr \mid a \notin X \} \\
\mathcal{RT}_{BTL}[\![\bigcirc\phi]\!] = \{ \langle\rangle, \langle\Sigma\rangle \} \cup \{ \langle X, a\rangle \frown tr \mid a \notin X \wedge tr \in \mathcal{RT}_{BTL}[\![\phi]\!] \} \\
\mathcal{RT}_{BTL}[\![\psi \wedge \phi]\!] = \mathcal{RT}_{BTL}[\![\psi]\!] \cap \mathcal{RT}_{BTL}[\![\phi]\!]
\end{array}$$

According to our translation function $pattern\ t \rightarrow P = event(t) \wedge \bigcirc(pattern\,P)$, it is easy to show that

$$\begin{array}{ll}
\mathcal{RT}_{BTL}[\![event(t) \wedge \bigcirc(pattern\,P)]\!] & \\
\quad = \mathcal{RT}_{BTL}[\![event(t)]\!] \cap \mathcal{RT}_{BTL}[\![\bigcirc(pattern\,P)]\!] & [\text{def of } \wedge] \\
\quad = \{ \langle\rangle \} \cup \{ \langle X, a\rangle \frown tr \mid a = event(t) \wedge a \notin X \wedge tr \in \mathcal{RT}_{BTL} \} & \\
\qquad \cap\, \mathcal{RT}_{BTL}[\![\bigcirc(pattern\,P)]\!] & [\text{def of } event(t)] \\
\quad = \{ \langle\rangle \} \cup \{ \langle X, a\rangle \frown tr \mid a = event(t) \wedge a \notin X \wedge tr \in \mathcal{RT}_{BTL} \} & [\text{def of } \bigcirc] \\
\qquad \cap\, \{ \langle X, a\rangle \frown tr \mid a \notin X \wedge tr \in \mathcal{RT}_{BTL}[\![pattern(P)]\!] \} \cup \{ \langle\rangle, \langle\Sigma\rangle \} & \\
\quad = \{ \langle X, a\rangle \frown tr \mid a = event(t) \wedge a \notin X \wedge tr \in \mathcal{RT}_{BTL}[\![pattern(P)]\!] \} & \\
\qquad \cup\, \{ \langle\rangle \} & [\text{def of } \cap] \\
\quad \supset \mathcal{RT}_{SPL}[\![t \rightarrow P]\!] &
\end{array}$$

Since this sub-language is used to describe behaviour inside a property specification and hence we only need to concentrate on finite refusal traces of the same length, subset inclusion will suffice.

The nondeterministic interleaving operator $\sqcap\!\sqcap$ is sequentialised by the function *npar* before being mapped into its CSP's equivalent. This function essentially implements the step law of $\sqcap\!\sqcap$ above via the function *initials* below and is defined as follows, where $P, Q \in SPL$.

$$\begin{array}{l}
npar(End, End) = End \\
npar(End, Q) = Q \\
npar(P, End) = P \\
npar(P, Q) = (\sqcap(a, X) : initials(P) \bullet a \rightarrow npar(X, Q)) \\
\qquad\qquad\qquad \sqcap\ (\sqcap(a, X) : initials(Q) \bullet a \rightarrow npar(X, P))
\end{array}$$

Similar to CSP [7], we write $\sqcap\, i : I \bullet P(i)$ to denote the nondeterministic choice of a set of indexed terms $P(i)$ where i ranges over I. The function *initials* takes a *SPL* model and returns a set of pairs, each pair contains a possible initial task enactment and the model after enacting that task. For example hp takes $a \rightarrow A \sqcap b \rightarrow B$ and returns the set $\{ (a, A), (b, B) \}$.

$$\begin{aligned}
&\mathit{initials}(P \sqcap Q) = \mathit{initials}(P) \cup \mathit{initials}(Q) \\
&\mathit{initials}(P \sqcap\!\sqcap Q) = \mathit{initials}(\mathit{npar}(P, Q)) \\
&\mathit{initials}(a \rightarrow P) = \{ (a, P) \} \\
&\mathit{initials}(\mathit{End}) = \emptyset
\end{aligned}$$

Going back to the example in Figure 2(b), we are now able to specify the pattern of behaviour $(a \rightarrow \mathit{End}) \sqcap\!\sqcap (b \rightarrow \mathit{End}))$ which states that tasks A and B are executed in parallel without needing to know their timing constraints. Here the BTL formula ϕ describes this pattern of behaviour:

$$\phi = (a \wedge \bigcirc b) \vee (b \wedge \bigcirc a)$$

and Spec is the corresponding CSP process of ϕ. We use event a to associate with some task A.

$$\begin{aligned}
\mathit{Spec} = \;&\mathbf{let} \\
&\quad \mathit{Spec}0 = b \rightarrow \mathit{Spec}2 \quad \mathit{Spec}1 = a \rightarrow \mathit{Spec}3 \\
&\quad \mathit{Spec}2 = a \rightarrow \mathit{Spec}4 \quad \mathit{Spec}3 = b \rightarrow \mathit{Spec}4 \\
&\quad \mathit{Spec}4 = \mathit{Stop} \sqcap (\textstyle\sqcap x : \Sigma \bullet x \rightarrow \mathit{Spec}4) \\
&\mathbf{in}\;\; \mathit{Spec}0 \sqcap \mathit{Spec}1
\end{aligned}$$

This allows us to make the following kinds of refinement assertions under the refusal traces semantics, where the implementation process may represent the behaviour under the timed model and the untimed model respectively.

$$\mathit{Spec} \sqsubseteq_{\mathcal{RT}} a \rightarrow b \rightarrow \mathit{Stop} \qquad \mathit{Spec} \sqsubseteq_{\mathcal{RT}} a \rightarrow \mathit{Stop} \;|||\; b \rightarrow \mathit{Stop}$$

4 Property Patterns

To assist the specification of behavioural properties in terms of the generalised property patterns, we define a property specification language PL by the following grammar:

$$\begin{aligned}
x, y \in \mathit{PL} &::= \texttt{Abs}(p, s) \mid \texttt{Un}(p, s) \mid \texttt{Ex}(p, n, s) \mid \texttt{BEx}(p, b, s) \mid \\
&\quad\;\; x \vee y \mid x \wedge y \qquad \textbf{where}\, p \in \mathit{SPL};\; n \in \mathbb{N};\; b \in \mathit{BL};\; s \in \mathit{SL} \\
\mathit{BL} &::= \;\leq n \mid \;= n \mid \;\geq n \quad \textbf{where}\, n \in \mathbb{N} \\
\mathit{SL} &::= \texttt{always} \mid \texttt{before}\,(p, n) \mid \texttt{after}\, p \mid \qquad \textbf{where}\, p \in \mathit{SPL};\; n \in \mathbb{N} \\
&\quad\;\; \texttt{between}\, p\, \texttt{and}\,(q, n) \mid \texttt{from}\, p\, \texttt{until}\,(q, n)
\end{aligned}$$

where each term in PL represents a behavioural property with respect to the property pattern, each term specifies the behavioural constraints over some *bounded*, *nondeterministic* behaviours specified by the sub-language SL. Throughout this section we use the term *state* in the sense of a transition system of a CSP process describing a BPMN diagram: a graph showing the states it can go through and actions, each denoted by a single CSP event, that it takes to get from one to another. Algebraically this is where each transition between states is an application of a step law. We describe each term in PL briefly as follows:

- $\texttt{Abs}(p, s)$ (Absence) states that the pattern of behaviour p must be refused throughout the scope s;
- $\texttt{Un}(p, s)$ (Universality) states that the pattern of behaviour p must occur throughout the scope s;
- $\texttt{Ex}(p, n, s)$ (Existence) states that the pattern of behaviour p must occur *at least once* during the scope s. In *LTL* one might model this property using the *eventually* operator; however as discussed earlier, it is not possible to model unbounded *eventually* specification, therefore we restrict this pattern with a bound and instead state that p must occur *at least once* within the subsequent n states from the start of scope s;
- $\texttt{BEx}(p, b, s)$ (Bounded Existence) states that the pattern of behaviour p must occur *a specified number of times*, defined by the bound b, throughout the scope s. A bound may either be *exactly* $(= n)$, *at least* $(\geq n)$ or *at most* $(\leq n)$;

Each property may be specified within one of the five different types of scope, which are captured by our sub-language *SL*. Here we describe each one briefly.

- $\texttt{always}$ (Global) states that the property in question must hold throughout all possible execution. For example $\texttt{Abs}(a \vee b, \texttt{always})$ states that both events a and b must be refused in all possible executions;
- $\texttt{before}\,(p, n)$ (Before p) states that if there exists the pattern of behaviour p in the subsequent n states, the property in question must hold before p for all possible executions. For example $\texttt{Un}(\texttt{available}\,a, \texttt{before}\,(b, n))$ states that a must not be refused before an occurrence of b in the subsequent n states.
- $\texttt{after}\,p$ (After p) states that if there exists the pattern of behaviour p in any one of the subsequent states from the start of the execution, then the property in question must hold precisely after that state. For example $\texttt{BEx}(a \vee b, \leq m, \texttt{after}\,c)$ states that a sequence of at most m as and bs must occur after the occurrence of the event c.
- $\texttt{between}\,p\,\texttt{and}\,(q, n)$ (Between p and q) states that if there exists an occurrence of some pattern of behaviour p that is succeeded by some other pattern of behaviour q in n subsequent states after p, then the property in question must hold after p and before q.
- $\texttt{from}\,p\,\texttt{until}\,(q, n)$ (After p until q) states that if there exists an occurrence of some pattern of behaviour p then the property in question must hold after p or if there exists an occurrence some pattern of behaviour q in the subsequent n states after p then the property in question must hold between p and q. Note that q does not ever have to occur.

Note *PL*'s grammar does not include the patterns such as *Precedence* or *Response* [1]; we do not see this as a shortcoming, as these patterns, belonging the set of *order* patterns, may be expressed in terms of *generalised* existence patterns where each property is over a set of patterns of behaviours. For convenience we define the function *next* such that $next_{\phi}\psi$ returns ψ composed with n next operators where n is the largest number of subsequent states about which ϕ make an assertion. For example the furthest state of the expression $a \vee b$ is 1; for both expressions $\bigcirc b$ and $a \wedge \bigcirc\, \texttt{available}\, c$ it is 2. It is not difficult to calculate the number of states a pattern of behaviour spans, as *SPL* is characterised by $\vee$, $\wedge$ and $\bigcirc$ operators over atomic formulae in *BTL*. The function *next* is defined as the functional composition ($\mathit{nexts} \circ \mathit{states}$) where $\mathit{states}(\phi)$ returns one minus the furthest state the expression ϕ, translated from some pattern of behaviour in *SPL*, specifies. The function *nexts* is defined such that $\mathit{nexts}_n \phi$ returns a composition of ϕ with n next operators, assuming $\mathit{nexts}_0 \phi = \phi$. We write the predicate *single* such that some *BTL* expression μ satisfies it, denoted as $\mathit{single}(\mu)$, if and only if μ specifies behaviours for only a single state; we say such expressions are *single state specifications.*

Also, we extend the grammar of *BTL*, denoted as BTL^δ, with the two derived temporal operators $\Diamond$ and $\widetilde{\mathcal{U}}$ to express *bounded eventuality* and *bounded until.* Since $\Diamond_n\, \phi = \texttt{true}\, \widetilde{\mathcal{U}}_n\, \phi$, we only define the semantics of $\widetilde{\mathcal{U}}$ as follows:

$$P \models \psi\, \widetilde{\mathcal{U}}_n\, \phi \equiv \forall\, tr : \mathcal{RT}[\![P]\!] \bullet \exists\, i : 0 \,..\, n \bullet \forall\, j : 0 \,..\, (i-1) \bullet \\ tr^i \in \mathcal{RT}_{BLT}[\![\phi]\!] \wedge tr^j \in \mathcal{RT}_{BLT}[\![\psi]\!]$$

where $1 \leq n < \#tr$ and we write tr^i for refusal trace tr with the first i events and i refusals removed for i ranging over the length of tr. We write $P \models \psi$ if every execution of process P satisfies the formula ψ. The following is the derivation of $\widetilde{\mathcal{U}}$ using operators in *BTL*:

$$\psi\, \widetilde{\mathcal{U}}_n\, \phi = \big(\bigwedge_{i \in \{\, 0..n-2\,\}} \mathit{nexts}_{i * \mathit{states}(\psi)}(\phi \vee \psi)\big) \wedge \mathit{nexts}_{(n-1) * \mathit{states}(\psi)} \phi \tag{3}$$

For example the formula $\Diamond_2\, (a \vee b)$ states that either task a or b must be performed at least once in the next two subsequent states; the corresponding formula in *BTL* is $(a \vee b) \vee (\texttt{true} \wedge \bigcirc(a \vee b))$. We write $\phi \Rightarrow \psi$ as a shorthand for $\neg\phi \vee (\phi \wedge \psi)$ where ϕ and ψ are expressions in BTL^δ and ϕ does not include operators $\Box$ and $\mathcal{R}$.

To assist our translation we define the partial function *negate* such that $\mathit{negate}(\phi)$ negates the formula ϕ by distributing the negation operator over temporal operators except the always ($\Box$) and the release ($\mathcal{R}$) operators. This is sufficient as the function is only applied to patterns of behaviour described in *SPL*, and we have shown in Section 3 that *SPL* can be completely characterised by $\wedge$ and $\bigcirc$ operators over atomic formulae in *BTL*. Here we only provide the partial definition of *negate*, where $\phi, \psi \in BTL^\delta$ and $n \in \mathbb{N}$, omitting the more trivial part of the definition.

$$\begin{aligned}
&\mathit{negate}(\phi \Rightarrow \psi) = \phi \wedge (\mathit{negate}(\phi) \vee \mathit{negate}(\psi)) \\
&\mathit{negate}(\Diamond_n\, \phi) = (\mathit{negate} \circ \mathit{derive})(\Diamond_n\, \phi) \\
&\mathit{negate}(\psi\, \widetilde{\mathcal{U}}_n\, \phi) = (\mathit{negate} \circ \mathit{derive})(\psi\, \widetilde{\mathcal{U}}_n\, \phi) \\
&\mathit{negate}(\bigcirc\phi) = \bigcirc(\mathit{negate}(\phi))
\end{aligned}$$

The function *derive* converts an expression in BTL^δ back to *BTL* according to the definition given in equation 3. The full definition may be found in the technical report version of this paper [10].

We define a translation function *makeTL* to be the functional composition ($derive \circ tl'$) that takes a property specification in PL and returns its corresponding temporal logic expression in BTL. The definition of tl' is as follows:

$$\begin{array}{ll} tl'(\sigma \wedge \rho) = tl'(\sigma) \wedge tl'(\rho) & tl'(\sigma \vee \rho) = tl'(\sigma) \vee tl'(\rho) \\ tl'(\texttt{Abs}(\mu, s)) = absence(\mu, s) & tl'(\texttt{Ex}(\mu, n, s)) = exist(\mu, n, s) \\ tl'(\texttt{BEx}(\mu, b, s)) = boundexist(\mu, b, s) & tl'(\texttt{Un}(\mu, s)) = universal(\mu, s) \end{array}$$

where $n \in \mathbb{N}$, $\mu, \nu \in SPL$, $b \in Bound$, $s \in SL$, and $\sigma, \rho \in PL$. Table 1 shows BTL^{δ} mappings of functions *absence*, *exist*, *universal* and *boundexist*. We assume p is the BTL expression of the pattern of behaviour μ, which we are interested in, and both $q = pattern(\nu)$ and $r = pattern(\upsilon)$. As a shorthand we write $\neg p$ for some patterns of behaviour of p to represent the negation of p by distributing $\neg$ as described by the function *negate*. For reasons of space we have chosen to describe only the formalisation of the *bounded existence* pattern and its corresponding function *boundexist*. Explanations for other patterns may be found in the longer technical report version of this paper [10].

4.1 Bounded Existence

The function *boundexist* takes a pattern of behaviour μ, a bound b and a scope s and returns the corresponding expression in BTL^{δ} stating μ must occur for the number of times specified by b within s and other behaviours may also within s. For reasons of space we assume every possible sequence of events defined by μ covers the same number of states, that is the *maximum* number of states. The longer technical report version of this paper [10] gives a complete formal treatment where this assumption is relaxed. Our definition also reflects the impossibility of expressing the *unbounded eventually* operator under the refusal traces model. We first define the function *bound* such that $bound(p, q, b)$ returns the corresponding expression in BTL^{δ} stating a bounded existence of behaviour p with no scope. Table 1 lists BTL^{δ} mappings for *bound*. Here we describe the formalisation for each type of bounds.

The expressions to model exactly n $(= n)$ occurrences of behaviour p may be written as $\bigwedge_{i \in \{0..n-1\}}(nexts_{i*states(p)}\, p) \wedge nexts_{n*states(p)}\,(q\, \mathcal{R}\, \neg p)$. Note since it is not possible to model unbounded eventually, and hence unbounded until operator, we restrict this pattern with all the occurrences of p occurring consecutively. This is not a problem as it is always possible to conceal all the other behaviours within the diagram in question via the CSP hiding operator. The condition $q\, \mathcal{R}\, \neg p$ is to ensure that p may not occur until some other behaviour q occurs, signifying the start of the pattern's scope. It is `false` if the scope is global. The expression to model at least n $(\geq n)$ occurrences of behaviour p may be written as $\bigwedge_{i \in \{0..n-1\}}(nexts_{i*states(p)}\, p)$. Since the bound is greater than or equal, the condition $q\, \mathcal{R}\, \neg p$ is not required. The expressions to model at most n $(\leq n)$ occurrences of behaviour p may be written as $nexts_{n*states(p)}\,(q\, \mathcal{R}\, \neg p)$. This expression states that any of the n instances of p may or may not occur and after which behaviour p may not occur until some other behaviour q occurs.

We now provide a description of our formalisation similar to the format when describing the absence pattern, assuming $p = pattern(\mu)$, $q = pattern(\nu)$ and $r = pattern(\upsilon)$. We write $getbound(b)$ for some bound b to denote the number part of the value.

- The global existence μ with bound b is modelled trivially as $bound(p, \mathtt{false}, b)$;
- The existence of μ with bound b before some behaviour ν is modelled as

  $$\diamondsuit_n\ q \Rightarrow \neg q\ \widetilde{\mathcal{U}}_{n-getbound(b)*states(p)}\ bound(p, q, b)$$

  which states that if ν occurs in one of the subsequent n states, then ν may only occur after the bounded number of μ occurs within the subsequent $n - getbound(b) * states(p)$ states;
- The existence of μ with bound b after some behaviour ν is modelled as $\Box(q \Rightarrow next_q(bound(p, q, b)))$, which states that if ν occurs at all then the bounded number of μ occurs immediately after ν;
- The existence of μ with bound b between behaviour ν and υ is modelled as

  $$\Box(q \Rightarrow (next_q\ \diamondsuit_n\ r \Rightarrow (bound(p, r, b) \wedge bound(p, r, b)\,\mathcal{R}\,\neg r \wedge r\,\mathcal{R}\,\neg q)))$$

  which states if the behaviour ν occurs and there exists some behaviour υ in the n subsequent states after ν has occurred, then υ cannot occur until a bounded number of instances of μ occur after ν occurs. Here n must be strictly larger than $getbound(b) * states(p)$, and we restrict this pattern so ν may only occur again after υ has occurred;
- The existence of μ after behaviour ν until υ is modelled as

  $$\Box(q \Rightarrow (next_q \neg r\ \widetilde{\mathcal{U}}_1\ bound(p, r \vee q, b)))$$

  which states if the behaviour ν occurs then either the bounded number of instances of μ must occur immediately after ν occurs. While the behaviour υ may not occur before the instances of μ have occurred, υ could occur after.

For example we could use the pattern "The bounded existence of μ after ν" to describe the property that either task A or C has to occur followed by either one of them again after Task B has occurred. This may be expressed in PL as $\mathtt{BEx}(a \rightarrow End \sqcap c \rightarrow End, = 2, \mathtt{after}\, b \rightarrow End)$, where a,b and c correspond to tasks A, B and C. The following is the CSP specification translated from the corresponding BTL expression,

$$\begin{array}{l}
Spec = \mathbf{let} \\
\qquad Spec0 = Proceed(\{\, b \,\}, Spec0 \sqcap Spec1) \\
\qquad Spec1 = b \rightarrow (Spec2 \sqcap Spec3) \quad Spec2 = c \rightarrow (Spec4 \sqcap Spec5) \\
\qquad Spec3 = a \rightarrow (Spec4 \sqcap Spec5) \quad Spec4 = c \rightarrow (Spec7 \sqcap Spec6) \\
\qquad Spec5 = a \rightarrow (Spec7 \sqcap Spec6) \quad Spec6 = b \rightarrow (Spec2 \sqcap Spec3) \\
\qquad Spec7 = Proceed(\{\, a, b, c \,\}, Spec7 \sqcap Spec6) \\
\quad \mathbf{in} \quad Spec0 \sqcap Spec1
\end{array}$$

where the parameterised process $Proceed$ is defined as follows:

$$Proceed(X, P) = Stop \sqcap Skip \sqcap (\sqcap\, x : \Sigma \setminus X \bullet x \rightarrow P)$$

4.2 Revisiting the Example

Going back to our running example, we may use the absence pattern "the absence of μ between some behaviours ν and υ" to specify the property of the travel agent. We denote task *Book_Seat* by event *bookseat*, and similarly for *Request_Cancel*, *Request_Timeout* and *Send_Invoice*; the following is the corresponding *PL* expression specifying this property.

$$\texttt{Abs}(\mathit{Cancel}, \texttt{between}\, \mathit{bookseat} \rightarrow \mathit{End}\, \texttt{and}(\mathit{sendinvoice} \rightarrow \mathit{End}, 2))$$

where the behaviour *Cancel* is defined as follows:

$$\mathit{Cancel} = \mathit{requestcancel} \rightarrow \mathit{End} \sqcap \mathit{reservetimeout} \rightarrow \mathit{End}$$

Here is the corresponding CSP process.

$$\begin{array}{l}
\mathit{Spec} = \textbf{let} \\
\quad \mathit{Spec}0 = \mathit{Proceed}(\{\, \mathit{bookseat} \,\}, \mathit{Spec}0 \sqcap \mathit{Spec}1) \\
\quad \mathit{Spec}1 = \mathit{bookseat} \rightarrow (\mathit{Spec}2 \sqcap \mathit{Spec}3 \sqcap \mathit{Spec}4 \sqcap \mathit{Spec}5 \sqcap \mathit{Spec}6) \\
\quad \mathit{Spec}2 = \mathit{Proceed}(\{\, \mathit{bookseat}, \mathit{sendinvoice} \,\}, \mathit{Spec}7 \sqcap \mathit{Spec}1) \\
\quad \mathit{Spec}3 = \mathit{sendinvoice} \rightarrow (\mathit{Spec}0 \sqcap \mathit{Spec}1) \\
\quad \mathit{Spec}4 = \mathit{bookseat} \rightarrow (\mathit{Spec}2 \sqcap \mathit{Spec}4 \sqcap \mathit{Spec}8 \sqcap \mathit{Spec}9) \\
\quad \mathit{Spec}5 = \mathit{Proceed}(\{\, \mathit{bookseat}, \mathit{requestcancel}, \mathit{reservetimeout} \,\}, \mathit{Spec}3) \\
\quad \mathit{Spec}6 = \mathit{bookseat} \rightarrow (\mathit{Spec}3) \\
\quad \mathit{Spec}7 = \mathit{Proceed}(\{\, \mathit{bookseat}, \mathit{sendinvoice} \,\}, \mathit{Spec}0 \sqcap \mathit{Spec}1) \\
\quad \mathit{Spec}8 = \textbf{let}\ \mathit{poss} = \{\mathit{bookseat}, \mathit{requestcancel}, \mathit{reservetimeout}, \mathit{sendinvoice}\} \\
\qquad\qquad\quad \textbf{in}\ \mathit{Proceed}(\mathit{poss}, \mathit{Spec}3) \\
\quad \mathit{Spec}9 = \mathit{bookseat} \rightarrow (\mathit{Spec}3) \\
\textbf{in}\quad \mathit{Spec}0 \sqcap \mathit{Spec}1
\end{array}$$

Now it is possible to see if the travel agent diagram satisfies this property by checking the following refusal traces refinement assertion using the FDR tool.

$$\mathit{Spec} \sqsubseteq_{\mathcal{RT}} \mathit{Agent} \setminus \Sigma \setminus \{\mathit{bookseat}, \mathit{requestcancel}, \mathit{reservetimeout}, \mathit{sendinvoice}\}$$

5 Conclusion

In this paper we considered the application of Dwyer et al.'s Property Specification Patterns for constructing behavioural properties, against which CSP models of BPMN diagrams may be verified. We proposed a property specification language *PL* for capturing the generalisation of the property patterns in which constraints are specified over patterns of behaviours rather than individual events. We then describe the translation from *PL* into a bounded, positive fragment of *LTL*, which can then be translated automatically into its corresponding CSP specification for simple refinement checks. We have demonstrated the application of our specification language via a couple of small examples. We have implemented a Haskell prototype of the translation[2], using Lowe's implementation. Our intention is to implement tool support allowing developers to build property specifications without the knowledge of *PL*, *LTL* or *CSP*.

[2] `http://www.comlab.ox.ac.uk/peter.wong/observation`

Acknowledgements

This work is supported by a grant from Microsoft Research. The authors would like to thank the anonymous referees for useful suggestions and comments.

References

1. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in Property Specifications for Finite-State Verification. In: Proceedings of the 21st International Conference on Software Engineering (1999)
2. Formal Systems (Europe) Ltd. Failures-Divergences Refinement, FDR2 User Manual (1998), http://www.fsel.com
3. Lowe, G.: Specification of communicating processes: temporal logic versus refusals-based refinement. Formal Aspects of Computing 20(3) (2008)
4. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Springer, Heidelberg (1992)
5. Mukarram, A.: A Refusal Testing Model for CSP. D.Phil thesis, University of Oxford (1992)
6. Object Management Group. BPMN Specification (February 2006), http://www.bpmn.org
7. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall, Englewood Cliffs (1998)
8. Wong, P.Y.H., Gibbons, J.: A Process Semantics for BPMN. In: Liu, S., Maibaum, T., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 355–374. Springer, Heidelberg (2008), http://www.comlab.ox.ac.uk/peter.wong/pub/bpmnsem.pdf
9. Wong, P.Y.H., Gibbons, J.: A Relative-Timed Semantics for BPMN. In: Proceedings of 7th International Workshop on the Foundations of Coordination Languages and Software Architectures, July 2008. ENTCS (2008); Invited for special issue in Science of Computer Programming, http://www.comlab.ox.ac.uk/peter.wong/pub/foclasa08.pdf
10. Wong, P.Y.H., Gibbons, J.: Property Specifications for Workflow Modelling. Technical Report, University of Oxford (2008), http://web.comlab.ox.ac.uk/oucl/work/peter.wong/pub/psp.pdf

Table 1. BTL^{δ} mapping of functions *absence*, *exist*, *universal*, *boundexist* and *bound*

	Universal (*universal*)	Absence (*absence*)
`always`	$\Box p$	$\Box\neg p$
`before`(ν, n)	$(\Box\neg q) \vee (p\,\widetilde{\mathcal{U}}_n\ q)$ or $(\Box\neg q) \vee (p \wedge nexts_n(q))$	$(\Box\neg q) \vee (\neg p\,\widetilde{\mathcal{U}}_n\ q)$
`after`ν	$\Box(q \Rightarrow (next_q\,(\Box\, p)))$ or $\Box(q \Rightarrow (next_q\, p))$	$\Box(q \Rightarrow (next_q\,(\Box\neg p)))$
`between`$\,\nu\,$`and`$\,(\upsilon, n)$	$\Box(q \Rightarrow next_q\,(\lhd\!\!\rhd_n\ r \Rightarrow (p\,\widetilde{\mathcal{U}}_n\ r)))$ or $\Box(q \Rightarrow next_q\,(\lhd\!\!\rhd_n\ r \Rightarrow (p \wedge nexts_n r)))$	$\Box(q \Rightarrow (next_q(\lhd\!\!\rhd_n\ r \Rightarrow (\neg p\,\widetilde{\mathcal{U}}_n\ r))))$
`from`$\,\nu\,$`until`$\,(\upsilon, n)$	$\Box(q \Rightarrow (next_q\,(\Box\, p \vee (p\,\widetilde{\mathcal{U}}_n\ r))))$ or $\Box(q \Rightarrow (next_q\,(p \vee nexts_n r)))$	$\Box(q \Rightarrow (next_q\,(\Box\neg p \vee (\neg p\,\widetilde{\mathcal{U}}_n\ r))))$

	Bounded Existence (*boundexist*)	Existence (*exist*)
`always`	$bound(p, \texttt{false}, b)$	$\lhd\!\!\rhd_n\ p$
`before`(ν, n)	$\lhd\!\!\rhd_n\ q \Rightarrow \neg q\,\widetilde{\mathcal{U}}_{n-getbound(b)*states(p)}\ bound(p, q, b)$	$\lhd\!\!\rhd_n\ q \Rightarrow (\neg q\,\widetilde{\mathcal{U}}_m\ p)$
`after`ν	$\Box(q \Rightarrow next_q(bound(p, q, b)))$	$\Box(q \Rightarrow (next_q\,(\lhd\!\!\rhd_m\ p)))$
`between`$\,\nu\,$`and`$\,(\upsilon, n)$	$\Box(q \Rightarrow (next_q(\lhd\!\!\rhd_n\ r \Rightarrow (bound(p, r, b) \wedge bound(p, r, b)\,\mathcal{R}\,\neg r \wedge r\,\mathcal{R}\,\neg q))))$	$\Box(q \Rightarrow next_q\,(\lhd\!\!\rhd_n\ r \Rightarrow (\neg r\,\widetilde{\mathcal{U}}_m\ p)))$
`from`$\,\nu\,$`until`$\,(\upsilon, n)$	$\Box(q \Rightarrow (next_q(\neg r\,\widetilde{\mathcal{U}}_1\ bound(p, r \vee q, b))))$	$\Box(q \Rightarrow next_q\,(\neg r\,\widetilde{\mathcal{U}}_m\ p))$

	BTL^{δ} mappings of *bound*
exactly n of p $(= n)$	$\bigwedge_{i\in\{\,0..n-1\,\}}(nexts_{i*states(p)}\ p) \wedge nexts_{n*states(p)}\ (q\,\mathcal{R}\,\neg p)$
at least n of p $(\geq n)$	$\bigwedge_{i\in\{\,0..n-1\,\}}(nexts_{i*states(p)}\ p)$
at most n of p $(\leq n)$	$nexts_{n*states(p)}\ (q\,\mathcal{R}\,\neg p)$

Formal Verification Based on Guided Random Walks

Thang H. Bui and Albert Nymeyer

School of Computer Science and Engineering,
The University of New South Wales, Australia
{buih,anymeyer}@cse.unsw.edu.au

Abstract. In software development, formal verification and simulation are seen as complimentary paradigms: the former can guarantee the correctness of systems with respect to properties, but does not scale; the latter does scale but cannot guarantee the absent of errors. In the authors' previous work, a mechanism of statically analysing a model has been used to build an abstraction of the original model, which in turn is used to guide a heuristic search in a guided model checker. We extend that work and apply the same technique to build a heuristically-driven, or guided, random-walk model checker. This work sits at the intersection of a number of research areas: model checking, random walks, heuristic search and simulation. Novel here is the use of a heuristic mechanism to guide the random walk towards states of the model that possibly violate user-defined properties, and the use of an automatic abstraction scheme to build the heuristic. In a series of experiments, we compare the performance of our guided, random-walk based tool to standard model-checking tools. A new metric that we call *Process Error Participation* (PEP) has also been devised to classify model behaviour.

1 Introduction

Verification, particularily model checking, is a technique used to guarantee the correctness of a model of a system with respect to a given desired property. In contrast, simulation is a technique that only partially verifies correctness. Being only partial, simulation is really just confidence building because in general no amount of simulation will guarantee correctness. Simulation in the form of testing dominates industrial development of both hardware and software. A reason for this is that no matter how large and complex a system is, it is always possible to carry out a simulation, unlike model checking, where users are inevitably confronted by time/space limitations and intractability. This comparison is simplistic however because it ignores a crucial task in system development, and that is the search to find errors. Verifiers are commonly used by system engineers to find errors even though their strength is to guarantee the absence of errors, unlike simulators, which are used to find errors and cannot guarantee their absence. The research outlined here seeks to bridge the gap between verification and simulation, and potentially provide a framework that will allow a system engineer to use the right tool at the right time: namely a tool to first debug a system and a related tool to verify its correctness, and do so seemlessly using the same model description and property specification.

A central concept in this research is *guided search*. Conventional model checkers such as SPIN [1] and NUSMV [2] use 'blind' (or fixed) search algorithms to traverse the state space and verify that each state satisfies some property. In recent years, a number of guided versions of conventional model checkers have been developed: for example Edelkamp et al. [3] who developed HSF-SPIN; Qian and Nymeyer [4,5] who developed GOLFER; Groce and Visser [6] who developed JPF and Kupferschmid et al. [7] who extended UPPAAL. Research in guided model checking is generally experimental because studying the effects of heuristics is necessarily empirical. In the worst case, a guided model checker will perform like a conventional model checker if the heuristic that acts as a guide has no effect. One method of adding guidance to a model checker is to use a search algorithm based on A*. Earlier work by the authors using this technique has resulted in GOLFER [8] and forms the backdrop to this research. GOLFER uses a heuristic generated by doing a static analysis of the model. The static analysis produces a data abstraction of the system, which is used to compute heuristic costs for each state of the system. The static analysis is based on the *cone of influence* abstraction technique [9]. The technique involves building an abstract model by computing the data dependencies in the concrete model and then removing 'weak' variables [4]. If the abstraction is well-informed, then the speed-up over conventional model checking can be dramatic [10].

Simulation has been used in model checking from the early days. Even today, model checkers like SPIN can be used as simulators. Simulation does not play any role in formal verification, although it will check assertions. Model-checking simulators based on *random walks* have also been the subject of recent research. The most notable and successful has been Owen et al. [11] who implemented a randomized search algorithm in a tool called LURCH. In trials, LURCH found 90% of the errors of SPIN and NUSMV, in orders of magnitude less time and space. Two basic problems beset users of random-walk based algorithms: when do you terminate a random walk if it does not terminate naturally, and when do you terminate a random-walk algorithm given complete coverage of the state space is generally not feasible? We address (but certainly do not solve) these problems in this research.

Seminal work in the area of random-walk based methods has been carried out by Grosu and Smolka [12] who were the first to present a formal *Monte Carlo* algorithm applied to model checking linear temporal logic (LTL). In that research, a property violation occurs if a so-called *lasso* (which is a random walk that ends in a cycle) containing an accept state is found in the Büchi automaton $B = B_S \times B_{\neg\varphi}$, where S is the system model and φ is the property. If no such lasso is found, Grosu and Smolka determine probabilistically how many random walks are required to achieve a level of certainty that the model satisfies the property. Note however, that it is not clear how practical or effective or even appropriate the probability argument is in limiting the search.

In this work, we extend earlier research [10,8,13,4,5] on symbolic guided model checking and add the capability of (guided) random-walk based search. The new algorithm uses the same symbolic heuristic to bias the random walks towards particular states that potentially violate user-defined properties. We also characterise models in terms of the number of processes that must 'participate' in an interaction that results in a state that violates a given property. We call this characterisation the *Process Error Participation* (PEP) factor. A low PEP factor means that few processes in the model are involved; a high PEP factor means that most or all are involved. PEP depends on both the user-defined property and the model: a deadlock may be caused by just two processes claiming a resource, or it may require all processes to claim the resource and halt progress. In seemingly related work [6], Groce and Visser study the so-called 'Thread Preference Heuristic'. However, their work is used to find heuristics for multi-threaded Java programs. Our heuristics are built by model abstractions, and the PEP factor is used to predict when substantial performance gains can be expected from particular model-checking approaches.

We will describe the relevant technical background in Sect. 2. Our algorithmic contribution is described in Sect. 3, where we define *random trails*, which form a subset of random walks. We present an algorithm to compute random trails and additionally, modify the algorithm to allow walks and trails to be *guided*. The experimental work is described in Sect. 4, including the comparison of various random-walk and random-trail based algorithms, both guided and unguided. Our conclusions and suggestions for future research are given in the last section.

2 Technical Background

2.1 Symbolic Abstraction-Guided Model Checking

Qian and Nymeyer [10,8,13,4,5] present a symbolic, guided model checking framework. The framework is implemented in a system called GOLFER, which in turn is based on NUSMV. Here we extract that portion of the underlying theory that is relevant to this research.

Definition 1 (Transition systems). *A finite state transition system is a tuple $M = (S, T, S_0, G)$, where S is a finite set of states, $T \subseteq S \times S$ is a transition relation, $S_0 \subseteq S$ is a set of initial states and $G \subseteq S$ is a set of goal states.*

In that work, a goal state is a state in which the checking properties are violated. The set of states of a transition system can be described by a non-empty set of state variables $X = (x_0, x_1, \dots, x_n)$, where each variable x_i ranges over a finite domain D_i. A homomorphic abstraction of a transition system is denoted by a set of surjections $H = (h_1, h_2, \dots, h_n)$, where each h_i maps a finite domain D_i to another domain $\hat{D}_i$ with $|\hat{D}_i| \leq |D_i|$. If we apply H to all states of a transition system, we generate an abstract version of the original, concrete system.

A homomorphic abstraction is a kind of relaxation of the concrete transition system in which certain groups of states are merged. Clarke et al. [14] show that a homomorphic abstraction preserves a class of temporal properties (namely ACTL*). In [4], the authors presented *Abstraction-Guided Model Checking* (AGMC), which uses the abstraction as a guide, or heuristic, in the model checker. The abstraction is used to calculate the shortest distance between each abstract state and the abstract goal state. It is proved in [8] that the distance between any state $\hat{s}$ and goal state $\hat{g}$ in the abstract system is a lower bound to the distance between its corresponding concrete states s and g. The abstract states and their corresponding distances are stored in a so-called *pattern database*, which provides a cost function to guide an A^*-based search algorithm towards the concrete goal state. If the search algorithm finds the goal state then the path it has found is guaranteed to be optimal.

The AGMC scheme in GOLFER is symbolic. In symbolic model checking, sets of states can be encoded as binary decision diagrams (BDDs). So for example, we represent the sets of abstract states that are equi-distant from the abstract goal state by a single BDD, denoted b_i, where i is the distance. In fact, abstract states and concrete states are encoded using the same set of state variables, with the difference that particular state variables in the abstract model are relaxed by giving them *don't care* status. No decoding is therefore required between abstract and concrete states. For comparison purposes, the AGMC algorithm described in [8] has been slightly modified, and renamed SGMC here. It is shown in Fig. 1. In particular we have removed those constructs in the AGMC algorithm that concern the symbolic implementation. We do this in all the symbolic algorithms shown in this work.

The SGMC algorithm maintains a set of current states *ss* called the *openSet* from which the algorithm must choose optimally. For each state in this set, the algorithm records the exact cost of reaching that state and the predicted cost h of reaching the goal from that state, also called the heuristic value. The *getMin* function, which extracts a set of states that has a minimum cost $(cost + h)$ from the input open set, is called each iteration. If it reaches a goal, the algorithm will return a path from an initial state to a goal state, which is called a *goal path*.

```
SGMC algorithm
Input: transition system $M = (S, T, S_0, G)$, abstractDB adb
Output: goal path | not found
 1: $openSet = \{(S_0, 0, 0)\}$
 2: $visitedStates = \emptyset$
 3: while $(openSet \neq \emptyset)$
 4:   $(ss, cost, h) = getMin(openSet)$
 5:   $visitedStates = visitedStates \cup ss$
 6:   return $buildPath(G, visitedStates)$ when $(ss \cap G \neq \emptyset)$
 7:   $ss' = \{s' \mid s \in ss, (s, s') \in T\} - visitedStates$
 8:   for all $b_i \in adb$
 9:     $ss'' = b_i \cap ss'$
10:     $openSet = openSet \cup \{(ss'', cost + 1, i)\}$ unless $(ss'' == \emptyset)$
11: return not found
```

Fig. 1. A Symbolic Abstraction-Guided Model Checking algorithm

The function *buildPath* extracts the goal path from *visitedStates*, starting at the goal states and working backwards to an initial state. The set of successor states ss' is computed in line 7. Those states in ss' that have already been visited are also removed. The set is then partitioned (line 8 to 10). Each partition is a set of states that are equi-distant from the goal, as given by the abstract pattern database, *adb*. The cost of a successor state is set to the cost of the parent state plus 1, and the state is added to *openSet*. Remembering that the algorithm is symbolic, the reader should note that an element in *openSet* is a tuple that consists of a BDD that encodes a set of states that are equi-distant from the goal state, together with the corresponding current and heuristic costs.

2.2 Random-Walk Based Model Checking

The simulation tool that we develop is based on the concept of *random walks*.

Definition 2 (Random walk). *A* random walk *in a transition system* $M = (S, T, S_0, G)$ *is a path* $\rho = s_0 s_1 \dots s_i s_{i+1} \dots s_n$ *in which the initial state* $s_0 \in S_0$*,* $s_i \in S$ *for* $1 \leq i \leq n$*, and the state* s_{i+1} *is chosen uniform randomly from the set of successors of* s_i *denoted* $\mathrm{Img}(s_i) = \{s \mid (s_i, s) \in T\}$*. A random walk is* accepted *if* $s_k \in G$*, where* $0 \leq k \leq n$*.*

A random-walk algorithm traverses the state space of a system using random walks, starting at an initial state. A random walk is accepted if it reaches the goal state, in which case the walk is a goal path. There are two important issues that need to be addressed: when do we terminate each random walk, and when do we terminate the algorithm? We refer to the former as the *end-walk condition*, and to the latter as the *termination condition*.

End-walk condition: A random walk may never end, so given finite time and memory, we need to determine some condition to end walks prematurely. In the research of Grosu and Smolka [12], the end of a walk occurs at the first loop. In other research [15,11], the maximum length of a walk is bound arbitrarily. In our research we consider random-walk algorithms that are bound and unbound.

Termination condition: Even if walks all end, we also need to consider how many walks will need to be carried out in the search for the goal state. In the research of Owen et al. [11], the maximum number of random walks is set arbitrarily. Another method involves tracking how many 'new' states are being visited relative to 'old' states and terminating when some *saturation point* is reached. Yet another method determines a termination condition probabilistically. It is used by Grosu and Smolka [12] in a tool called MC^2. Their work involved modelling the system S and (LTL) property φ as a Büchi automaton $B = B_S \times B_{\neg\varphi}$ and checking whether the language of this automaton $L(B)$ is empty. Given input parameters δ and ϵ, MC^2 takes $N = \ln(\delta)/\ln(1-\epsilon)$ random samples (which are random walks ending in a cycle, called *lassos*) from B to decide if $L(B) = \emptyset$. Should a sample discover an accepting lasso l, MC^2 returns false with l as a witness. Otherwise, it returns true. If we assume that the expectation of an accepting lasso is greater than ϵ, then upon further sampling (after N unsuccessful samples), the probability of finding a lasso is less than δ. This probabilistic termination condition can be applied only if the search space covers the probability space.

3 Random-Walk and Abstraction-Guided Algorithms

In this section we develop the random-walk and abstraction-guided algorithms that form the basis of this work. A random walk is defined too generally because of the presence of loops. We define a subset of random walks called *random trails.*

Definition 3 (Random trail). *A* random trail *in a transition system $M = (S, T, S_0, G)$ is a random walk $\theta = s_0 s_1 \dots s_n$ in which $\forall\, 0 \leq i \neq j \leq n$, $s_i \neq s_j$. A random trail is accepted if $s_k \in G$, $0 \leq k \leq n$. The* probability of a trail $\mathbf{Pr}[\theta]$ *is defined as:* $\mathbf{Pr}[s_0] = |S_0|^{-1}$ *and* $\mathbf{Pr}[s_0 s_1 \dots s_{n-1} s_n] = \mathbf{Pr}[s_0 s_1 \dots s_{n-1}] * |\mathrm{Img}(s_{n-1}) - \{s_0, s_1, \dots, s_{n-1}\}|^{-1}$.

Theorem 1. *A* maximal *random trail is a random trail that ends at a leaf state or a state in which all of its children have already been visited. Given a transition system $M = (S, T, S_0, G)$ and suppose that all states in S are reachable from some state in S_0, the set of all maximal random trails in S: (1) covers the entire state space; and (2) together with their probabilities, covers the probability space.*

Point (1) is trivial to prove; point (2) can be proved by induction on the number of states. Given input parameters δ and ϵ, and by considering maximal random trails only, we can terminate the random-search algorithm after executing $N = \ln(\delta)/\ln(1-\epsilon)$ random trails if no goal is found.

Some care is needed in the random-trail algorithm to record visited states because a trail is not allowed to revisit a state. In this research, states that have been visited in a trail are represented by a BDD, which is used to reduce (using BDD set operations) the set of potential successor states of the current state, which of course is also represented by a BDD. The actual successor state is (randomly) chosen from this set of states. Alternatively, in an explicit-state environment, a hash table is used to store and retrieve visited states (as in SPIN).

In a *guided* random walk, each state is assigned its minimum heuristic cost (i.e. distance) to a goal state. In this work, we use an abstraction of the system as a heuristic to assign a cost to each state, and use this cost to guide the supposedly random walk towards a possible goal. During a walk, states with smaller costs are chosen more often than states with larger costs. This abstraction heuristic mechanism is precisely the same approach used in AGMC [4]. In Fig. 2, we show the new symbolic abstraction-guided random-walk model-checking algorithm SGRMC. Remember, as in SGMC, the algorithm is actually symbolic, and the details of the BDD data structures and operations have been left out. The algorithm handles both walks and trails and both guided and unguided random walks. If the variable *guided* is false, the algorithm is a "pure" (i.e. unguided) random-walk algorithm, which for experimental purposes we call SRMC. Otherwise the algorithm will be guided, and is referred to as SGRMC.

```
SGRMC algorithm
Input: transition system $M = (S, T, S_0, G)$, abstractDB $adb$,
       boolean $guided$, boolean $randTrail$
Output: $goal\ path$ | not found
 1: while (! $terminationCond$)
 2:   $s = random(S_0)$
 3:   $visitedStates = \emptyset$
 4:   while (! $endCond(s)$)
 5:     $visitedStates = visitedStates \cup \{s\}$
 6:     return $getPath(visitedStates)$ when $(s \in G)$
 7:     $ss = \{s' \mid (s, s') \in T\}$
 8:     $ss = ss - visitedStates$ when $(randTrail)$
 9:     if $(guided)$
10:       costSet $p = \emptyset$
11:       for all $b_i \in adb$
12:         $ss'' = b_i \cap ss$
13:         $p = p \cup \{(ss'', i)\}$ unless $(ss'' == \emptyset)$
14:       $ss = guidedRandom(p, |adb|)$
15:     $s = random(ss)$
16: return not found
```

Fig. 2. A Symbolic Abstraction-Guided Random-Walk Model Checking algorithm

The algorithm SGRMC randomly and repeatedly traverses the state space of the given system until *terminationCond* is satisfied. Each traversal is restricted in length by *endCond*. In each traversal, all visited states are recorded (line 5) in *visitedStates*. The algorithm will return the goal path if a goal is located (lines 6). Otherwise, the algorithm will return **not found** when the termination condition is satisfied. The function *getPath* examines the times the states were visited and extracts the goal path. The set of successor states ss of a state s is generated in line 7. The input Boolean variable *randTrail* enables us to carry out random walks or random trails. If *randTrail* is true then we remove the visited states whenever they occur in the set of successor states (lines 8). The function *random* chooses a next state, uniformly and randomly, from ss (lines 2 and 15).

Like the SGMC algorithm, the algorithm divides ss into partitions, which are stored in a variable p that has type ***costSet*** (line 10 to 13). Each partition is assigned a cost i, where i is the abstract distance to the goals of all the states in that partition (line 13). A guided random-selection function, called *guidedRandom*, is called to compute the set of successor states ss (line 14). This function takes as input $p = \{(ss_0, i_0), \ldots, (ss_n, i_n)\}$ and the size of the input abstract database $|adb|$, and selects one of the sets ss_k from p in a biased fashion. The smaller the cost i_k, the greater the probability that ss_k will be selected. To achieve this, each instance ss_k is repeated $|adb| - i_k$ times, and then a set is selected randomly. The set of successor states is then restricted to that partition only. Thereafter, a successor state is selected uniformly and randomly from the (restricted) set of successor states (line 15).

4 Experimentation

In this section, we apply the four algorithms SRMC, SGRMC, SGMC and NUSMV to four well-known problems: the dining-philosophers problem, leader-election protocol, send/receive protocol and mutual-exclusion protocol[1]. Precisely the same (abstraction-based) heuristic is used in both guided algorithms SGMC and SGRMC. As we are interested in finding bugs, 'error' versions of each of the protocols that violate particular properties are used in this research.

Note that the user-defined properties are important in determining the PEP factor. It is easy to see that in the dining-philosophers problem, all processes need to 'work together' to result in deadlock, so its PEP factor is high. The leader-election protocol also has a high PEP factor because, during an election, all processes need to pass a token around the ring. In this research, our 'buggy' version allows two processes erroneously claiming a single resource. In the send/receive protocol that we use, a send process has a time-out (error) if a receive process does not send an acknowledgement on time, so just two processes are involved in an error. The PEP factor of this protocol is hence low. Similarly, in the mutual-exclusion protocol, we need just two processes to be in the critical section at the same time for an error to occur, so it also has a low PEP factor.

We note again that the PEP factor is dependent on the property that the user defines. So, for example, if the property in the mutual-exclusion protocol is that all processes cannot be pair-wise in the critical section at the same time, then the model is high PEP. We add that there is no relationship between the PEP factor and the tightness/looseness of the process coupling in the model.

In all experiment results in this section, each data point in the random-walk experiments is the average of executing the algorithm 50 times, where each execution uses a different seed in the random-number generator. All the experiments were performed on a Pentium IV 3.0GHz with 1GB RAM running a Linux operating system called Ubuntu 6.06.

4.1 Which Is the 'Fastest' Random-Walk Based Algorithm?

In this first series of experiments, we compare the 'unguided' random-walk SRMC_w, 'unguided' random-trail algorithm SRMC_t, and guided random-walk algorithm SGRMC_w, guided random-trail algorithm SGRMC_t.

4.1.1 Unbounded Random Walks and Trails

We first carry out the experiments with no bound on the length of the random walks and trails, and in the next section repeat the experiments with a bound.

Dining Philosophers Problem: In Fig. 3, we compare the experimental results of all algorithms for the dining-philosophers problem. In the table we show the number of philosophers **ph.**, execution **time** (in seconds), standard deviation of the time **std**, number of traversals to reach the goal **N** and length of the goal

[1] The first two of these problems can be found at http://anna.fi.muni.cz/benchmark.

ph.	SRMC$_w$				SRMC$_t$				SGRMC$_w$				SGRMC$_t$			
	time	std	N	len	time	std	N	len	time	std	N	len	time	std	N	len
10	0.32	0.3	1	361.6	0.30	0.3	1.12	285.2	0.19	0.1	1	215.7	0.17	0.1	1.06	155.1
12	0.89	0.7	1	602.8	0.83	0.5	1.04	531.3	0.66	0.4	1	467.0	0.43	0.3	1.02	284.9
14	1.98	1.6	1	968.8	1.65	1.2	1.06	783.5	0.96	0.7	1	471.7	0.92	0.7	1.06	375.3
16	5.78	5.0	1	2080.9	4.66	3.7	1.08	1532.6	2.75	1.9	1	975.6	2.02	1.5	1.16	564.4
18	12.34	12.0	1	3298.7	16.26	14.6	1.12	3870.6	6.20	4.8	1	1528.6	3.42	2.7	1.04	753.3

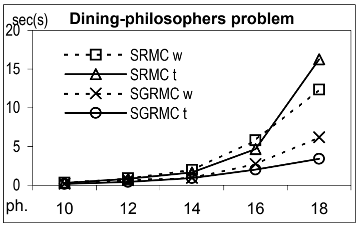
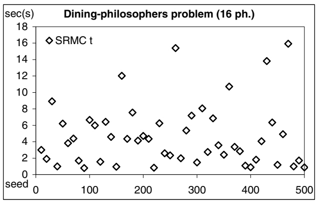


Fig. 3. Performance of unbounded random walks on the dining-philosophers problem



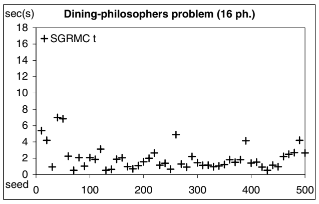


Fig. 4. Distributions of time for 16 dining philosophers using SRMC$_t$ and SGRMC$_t$

pr.	SRMC$_w$				SRMC$_t$				SGRMC$_w$				SGRMC$_t$			
	time	std	N	len	time	std	N	len	time	std	N	len	time	std	N	len
3	0.92	0.9	1	1584.3	0.81	0.7	1	936.8	0.42	0.4	1	685.2	0.33	0.3	1	333.9
4	4.67	4.1	1	4644.7	2.91	2.4	1	1566.7	3.02	2.8	1	3068.2	2.32	1.9	1	1206.1
5	29.85	31.0	1	19246.0	16.07	13.9	1	5262.1	13.61	13.2	1	8915.7	10.05	9.6	1	3025.8
6	176.94	217.1	1	69492.8	89.30	81.6	1	16664.5	106.75	96.6	1	42177.5	33.31	35.0	1	6126.3

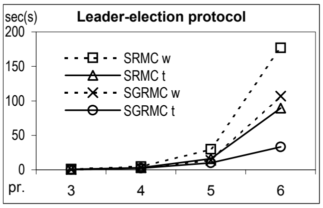


Fig. 5. Random walks for the leader-election protocol

path **len**. On the right of the table we plot the times. There are a number of observations that can be made: 1) The 'walk' algorithms reach the goal in just 1 traversal, while the 'trail' algorithms may need more than one. The reason is that trails must restart if all the successors of a state have been visited before. 2) The length of the goal path is much greater than optimal (note the minimum length is twice the number of philosophers). 3) On average, the fastest algorithm is the guided trail algorithm, SGRMC$_t$, which appears to scale well.

We also looked at the distribution of the data, and noting that the guided data had a lower standard deviation than the unguided data, found that the guided data was better behaved. In Fig. 4 we show for example the time taken to find the deadlock for 16 dining philosophers using SRMC$_t$ and using SGRMC$_t$.

Leader-Election Protocol: The other high PEP factor system is the leader-election protocol., whose results are shown in Fig. 5. The column **pr.** indicates the number of processes. The rest of the columns have the same meaning as before. We note the following: 1) The behaviour of the standard deviation of the

pr.	SRMC$_w$ time	std	N	len	SRMC$_t$ time	std	N	len	SGRMC$_w$ time	std	N	len	SGRMC$_t$ time	std	N	len
3	56.84	65.2	1	95589.4	32.55	42.4	3.7	7070.1	0.06	0.01	1	124.9	0.07	0.02	1	119.9
4	119.51	139.1	1	89829.8	43.67	61.9	1.1	25032.2	0.12	0.03	1	132.4	0.13	0.03	1	125.5
5	244.79	338.1	1	105878.6	124.81	130.7	1.0	46591.5	0.15	0.06	1	125.0	0.15	0.05	1	121.7
6	343.32	413.2	1	104177.0	250.56	270.7	1.0	68416.9	0.20	0.07	1	120.3	0.21	0.06	1	119.4

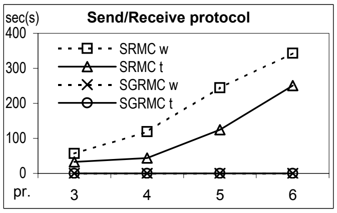


Fig. 6. Random walks for the send/receive protocol

pr.	SRMC$_w$ time	std	N	len	SRMC$_t$ time	std	N	len	SGRMC$_w$ time	std	N	len	SGRMC$_t$ time	std	N	len
16	0.22	0.21	1	111.9	0.20	0.16	1	107.6	0.12	0.06	1	41.3	0.13	0.06	1	41.3
20	0.29	0.26	1	93.9	0.31	0.29	1	97.5	0.20	0.09	1	42.8	0.20	0.08	1	42.0
24	0.55	0.46	1	123.0	0.53	0.42	1	114.7	0.23	0.11	1	30.3	0.24	0.12	1	30.3
28	0.54	0.42	1	82.5	0.55	0.44	1	85.2	0.34	0.20	1	35.5	0.34	0.20	1	35.5

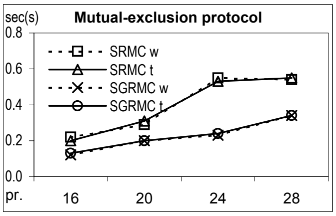


Fig. 7. Random walks for the mutual-exclusion protocol

time, and the number of traversals required have similar behaviour to the dining-philosophers problem. 2) The use of trails seems more effective than walks. 3) The guided trail algorithm SGRMC$_t$ was fastest and scaled the best.

Send/Receive Protocol: Unlike the previous two protocols, this protocol has a low PEP factor. The experimental results are shown in Fig. 6. The main observation here is that the guided walk and guided trail algorithms show similar behaviour, and are both much faster than the unguided algorithms. The conjecture is that the guide is more effective in a low PEP system because the costs of states associated with processes that play a role in the violation of the property are significantly lower than other costs of other states, so it is easier to generate a guide that is informed, and hence effective.

Mutual Exclusion Protocol: The results for this other low PEP factor system is presented in Fig. 7. The main observation here is that all algorithms perform well, although the guided algorithms perform better taking approximately half the time of the unguided algorithms. The reason for this is that there are many paths that lead to an error state and the error state is close by, so all search algorithms, guided or not, can quickly find the error state.

4.1.2 Bounded Random Walks and Trails

It is easy to speculate that an *unbounded* random walk through a state space (as all the experiments above are) can waste time 'going in the wrong direction'. In such cases, terminating the walk prematurely, and restarting it, may improve its chances of finding the error. However, the downside to this strategy is that if the bound is too short, we may never walk far enough to find the error (it may

(a)

ph.	SRMC$_w$	SRMC$_t$	SGRMC$_w$	SGRMC$_t$
10	0%	0%	0%	0%
12	-2%	-1%	2%	2%
14	7%	0%	6%	0%
16	22%	26%	14%	7%
18	41%	6%	-1%	4%

(b)

pr.	SRMC$_w$	SRMC$_t$	SGRMC$_w$	SGRMC$_t$
3	39%	28%	0%	-15%
4	186%	44%	96%	28%
5	640%	100%	522%	72%
6	182%	140%	623%	173%

(c)

pr.	SRMC$_w$	SRMC$_t$	SGRMC$_w$	SGRMC$_t$
3	-91%	-82%	0%	0%
4	-90%	-77%	-17%	-15%
5	-92%	-84%	13%	13%
6	-92%	-88%	5%	5%

(d)

pr.	SRMC$_w$	SRMC$_t$	SGRMC$_w$	SGRMC$_t$
16	0%	5%	0%	-8%
20	0%	-3%	0%	0%
24	0%	0%	0%	-4%
28	0%	0%	0%	0%

Fig. 8. Percentage change in execution time in the bounded (a) dining-philosophers problem, (b) leader-election protocol, (c) send/receive protocol, and (d) mutual-exclusion protocol

always stop short). There is hence a trade-off between the length and number of walks. To understand this trade-off better, we arbitrarily apply a bound to the length of all random walks. This bound is 100 times the number of concurrent processes in the system. The experiments are repeated with this bound.

In Fig. 8(a), we show the execution time for the **dining-philosophers problem**, expressed as a percentage change of the previous time. For example, consider the case with 18 philosophers and the unguided algorithm SRMC. The bound is set to 1800 steps, with the result that (on average) the search is 41% *slower* than the unbounded case. The main conclusion one can draw from these results is that arbitrarily bounding the walk/trail generally results in a worse execution time, but that the guided algorithms are less affected than unguided algorithms.

In Fig. 8(b) we see in the results for the **leader-election protocol** that the degradation in performance is even more pronounced, but that the trail algorithms are less affected. In the results for the **send/receive protocol** shown in Fig. 8(c), we see that the unguided walk algorithm's performance improves approximately 90% when we bound the walk (so e.g. the execution time for bound SRMC is approximately 27 seconds, which we note is still 100 times slower than SGRMC). Remembering from the unbounded experiment that the goal is only a short distance from the initial state, it is intuitively obvious that bounding the unguided algorithms is desirable to avoid long walks 'getting lost'. The guided algorithm already efficiently finds the error, so it gains little by being bound.

Finally we have the results for the **mutual-exclusion protocol** shown in Fig. 8(d). In this figure we see that bounding the walks and trails has virtually no effect on the time. What is important to note here is that, while this protocol and the send/receive protocol both have error states at only a short distance from the initial state, the number of processes in this protocol is much higher than the send/receive protocol (28 versus 6). This means that the bound for this protocol is much larger (remember that the bound is set to 100 times the number of processes), and a large bound is in effect no bound.

Discussion: These experiments suggest that unguided algorithms are faster with a bound than without if the bound is appropriate (i.e. not too long). Clearly, the bound should not be a function of just the number of processes but should take into account the structure of the state space and closeness of the error. In certain circumstances it is advantageous to apply a bound to random search (e.g. if the error is ‘close’ and the guiding abstraction is weak). However, any bound carries a risk of preventing the search from finding errors that are ‘too far away’.

Overall, the guided algorithms are significantly faster for high PEP systems, and orders of magnitude faster for low PEP systems. In particular, the guided trail algorithm SGRMC is almost always the fastest, and it finds the shortest path to the error state (but this path may still be longer than the optimal path, as we shall see in the next section). Comparing the unguided and guided random trail algorithms, we see that the guided random trail algorithm always results in a shorter path, which suggests that the guide is having a significant effect. It would be interesting to study the relationship between the size of this effect and the ‘quality’ of the heuristic. There did not seem to be any relationship between the size of the effect and the PEP factor.

4.2 How Much Faster Is (Guided) Simulation Than Verification?

In a second series of experiments, we compare the fastest simulation algorithm (SGRMC$_t$) with two verifiers: the symbolic model checker NUSMV and the symbolic abstraction-guided model checker SGMC. For the purposes of comparison, we consider just the unbounded version of SGRMC$_t$.

In Fig. 9, we show the comparison for the **dining-philosophers problem**. The table on the left of this figure shows the execution **time** (in seconds), the amount of memory used **mem** (in Megabytes) and the length of the goal path **len** for each algorithm. We also include the standard deviation **std** in the case of SGRMC$_t$. The chart on the right plots the execution times for up to 12 philosophers. While non-optimal, the results show that SGRMC$_t$ is orders of magnitude faster. This is somewhat a surprising result as it is well known that the ‘pathological’ high level of symmetry in the dining-philosophers problem is problematic for guided verification. This does not extend to guided simulation, even though it uses precisely the same heuristic. The reader may remember that even the unguided random-walk algorithm SRMC did very well on this model, so clearly simulation is a better approach than verification in this case.

ph.	NuSMV			SGMC			SGRMC$_t$			
	time	mem	len	time	mem	len	time	std	mem	len
4	0.01	0.76	8	0.01	0.87	8	0.01	0.00	0.88	31.80
6	0.06	1.70	12	0.12	2.47	12	0.03	0.01	1.25	68.60
8	0.56	8.82	16	1.78	11.46	16	0.07	0.02	1.82	91.90
10	11.91	11.41	20	54.26	15.15	20	0.17	0.07	3.29	155.10
12	106.75	40.58	24	168.50	37.49	24	0.43	0.27	6.65	284.90
14	882.94	203.69	28	15704.50	337.47	28	0.92	0.72	9.51	375.30
16	8693.00	886.83	32	> 10h			2.02	1.50	11.20	564.40
18	> 10h			> 10h			3.42	2.66	11.51	753.30
20	> 10h			> 10h			7.11	5.35	12.24	1038.10

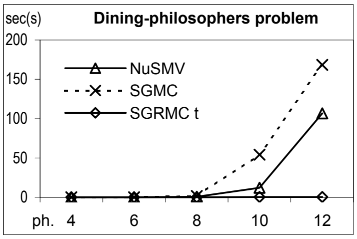


Fig. 9. Comparing simulation and verification for the dining-philosophers problem

pr.	NuSMV			SGMC			SGRMC $_t$			
	time	mem	len	time	mem	len	time	std	mem	len
3	0.74	10.00	18	0.48	6.86	18	0.33	0.30	4.36	333.86
4	20.37	22.93	19	12.56	11.58	19	2.32	1.87	10.16	1206.08
5	237.30	145.30	21	164.87	33.58	21	10.05	9.58	11.40	3025.82
6	2108.62	645.14	23	755.19	121.39	23	33.31	35.01	13.25	6126.34

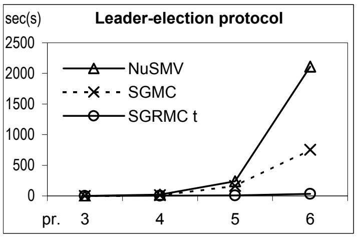


Fig. 10. Comparing simulation and verification for the leader-election protocol

pr.	NuSMV			SGMC			SGRMC $_t$			
	time	mem	len	time	mem	len	time	std	mem	len
5	2.60	11.46	13	0.07	2.08	13	0.04	0.05	3.47	121.70
10	26.93	24.54	13	0.16	3.79	13	0.60	0.20	9.17	124.36
15	72.47	49.01	13	0.27	5.11	13	1.49	0.60	9.49	140.44
20	150.97	105.51	13	0.55	8.44	13	2.69	0.96	9.69	136.12
25	273.65	152.72	13	1.08	9.73	13	4.63	2.25	10.20	132.16
30	421.76	225.04	13	1.35	8.34	13	8.25	5.49	11.06	158.26

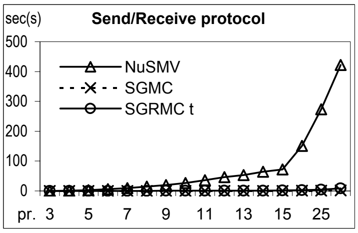
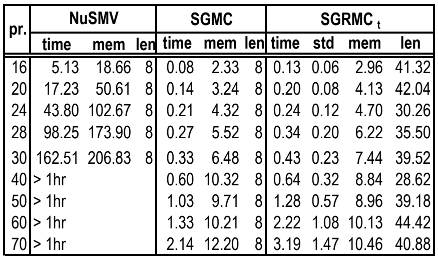


Fig. 11. Comparing simulation and verification for the send/receive protocol

The poor performance of guided verification (SGMC) is well-known and caused by the symmetry in the model that makes it difficult to find a useful heuristic. In effect, in this model SGMC works breadth-first like NUSMV, but has extra overhead due to 'expensive' BDD (partitioning) operations that it must perform (and NUSMV does not). Also interesting is the efficient memory utilisation of SGRMC$_t$, remembering this algorithm is based on BDDs.

The graph of the performance of the other high PEP factor model, the **leader-election protocol**, is shown in Fig. 10. The guided random-trail algorithm SGRMC$_t$ is again faster, and uses less memory, than the verification algorithms. However, comparing the verification algorithms, SGMC performed better than NUSMV. The reason for this is the system does not have as high a PEP factor as the dining-philosophers problem. While all processes in this system need to be involved in passing a message around the ring, in our system, a process may unilaterally claim leadership. Because other processes will be unaware of this, an error can result. This lower PEP-factor behaviour allows the heuristic to be a more effective guide, resulting in significantly better performance.

In the random-walk experiments we saw that both the guided random-walk and random-trail algorithms were particularly effective in low PEP factor systems. In Fig. 11 we compare verification and simulation for the **send/receive protocol**. The graph on the left in this figure shows that both guided verification and simulation are very fast compared to NUSMV, and both require far less memory. It is conjectured that the asymmetry in the search space caused by the relatively few processes play a role in the error state makes guiding effective. Interestingly, guided verification has better performance than guided simulation;



pr.	NuSMV			SGMC			SGRMC $_t$			
	time	mem	len	time	mem	len	time	std	mem	len
16	5.13	18.66	8	0.08	2.33	8	0.13	0.06	2.96	41.32
20	17.23	50.61	8	0.14	3.24	8	0.20	0.08	4.13	42.04
24	43.80	102.67	8	0.21	4.32	8	0.24	0.12	4.70	30.26
28	98.25	173.90	8	0.27	5.52	8	0.34	0.20	6.22	35.50
30	162.51	206.83	8	0.33	6.48	8	0.43	0.23	7.44	39.52
40	> 1hr			0.60	10.32	8	0.64	0.32	8.84	28.62
50	> 1hr			1.03	9.71	8	1.28	0.57	8.96	39.18
60	> 1hr			1.33	10.21	8	2.22	1.08	10.13	44.42
70	> 1hr			2.14	12.20	8	3.19	1.47	10.46	40.88

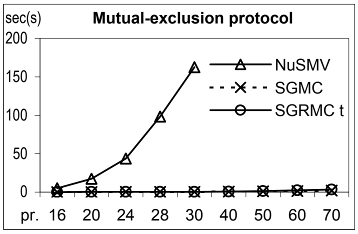


Fig. 12. Comparing simulation and verification for the mutual exclusion protocol

not only in execution time but also in memory usage. Contributing to the worse performance of simulation is undoubtedly the long path it takes to reach the error state. As expected, both NUSMV and SGMC generate optimal goal path lengths of course, while SGRMC$_t$ does not by an order of magnitude.

The experimental results for the other low PEP factor system, the **mutual-exclusion protocol**, are shown in Fig. 12. The same general behaviour described above is seen here: the guided algorithms SGMC and SGRMC$_t$ are both many orders of magnitude faster than (unguided) NUSMV, and require far less memory. However, the memory requirements of both algorithms are similar this time. As before, the verification algorithms are optimal, simulation is not.

Discussion: As expected, the guided random simulator SGRMC$_t$ is many orders of magnitude faster and uses less memory than the conventional verifier NUSMV in the series of experiments carried out here. In fact, SGRMC$_t$ can find errors in models that are intractable for NUSMV.

A comparison with the other verifier, SGMC, is quite different. Surprisingly, for low PEP factor systems, guided verification (i.e. SGMC) is faster than guided simulation (SGRMC). The reason for this is that, although precisely the same heuristic is used in guided verification and simulation, it plays a different role in each. In the former, there is no randomness. In the latter, the walk is still predominately random.

Of course, if successful, the guided verifier SGRMC generates an optimal goal path, and guided simulation does not. So, while a random-trail algorithm can find an error state very quickly, the path to the error may be long and circuitous, and one would assume unhelpful to the user unless the user had a tool, for example, that could prune the path.

5 Future Work and Conclusions

In this work we have developed symbolic, guided random-walk and random-trail model checking algorithms, applied these algorithms to four well-known problems, investigated the effect of bounding the length of walks and trails, and compared the fastest of these algorithms with conventional and guided symbolic model checking algorithms. We will not repeat the discussions here of the experimental results from Sect. 4.1 and 4.2, except to say that the guided random-trail algorithm SGRMC$_t$ is generally the best at locating errors, and when it is not, the guided model checker SGMC is the best. Each of the protocols studied in this work contained an error, and the error was always found. We did not consider what conclusions can be drawn if no error was found.

Noting that the guided random-walk approach uses precisely the same abstract model as heuristic as the guided model checker, a tool that allows the verification engineer to alternately and seamlessly carry out property-based simulation and verification is possible. What is necessary here is a 'feedback' loop that would integrate the two technologies and allow the verifier to benefit from the information obtained by the simulator.

Pelánek [16] has shown the structure, or shape, of state spaces for many specifications. The PEP characterisation that we have used is an attempt to relate these structures and properties. A low PEP factor can be viewed as indicating asymmetry in the state space. The more 'skew' the structure is w.r.t. the property, the easier it is to create an abstract model that will act as an effective heuristic. However, there is much work that needs to be done to understand the PEP factor. Can it be formalised for example?

Another aspect that needs to be considered in the method used to generate the abstract model. Currently this is done by using a 'crude' static analysis of the specification. While it ranks variables in terms of their dependencies, it does not, for example, rank variables with respect to the property under investigation, which could be important. Also not addressed in this work is the symbolic aspect. While all algorithms have been implemented using BDDs, the benefits (if any) of this approach have not been considered. A symbolic approach can be effective in model verification. It may or may not be in model simulation.

Finally, only small protocols have been studied in this work. The application of guided random walks to more realistic industrial case studies has yet to be carried out. The expectation however is that in real life, heuristics can be very effective because small but identifiable parts of the specification considered with respect to the property under investigation generally dominate the behaviour and lead to states that violate the property.

References

1. SPIN: on-the-fly, LTL model checking, `http://spinroot.com/spin/`
2. NuSMV: a new symbolic model checker, `http://nusmv.irst.itc.it/`
3. Edelkamp, S., Lluch Lafuente, A., Leue, S.: Directed explicit model checking with HSF-SPIN. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 57–79. Springer, Heidelberg (2001)
4. Qian, K., Nymeyer, A., Susanto, S.: Abstraction-guided model checking using symbolic IDA* and heuristic synthesis. In: Wang, F. (ed.) FORTE 2005. LNCS, vol. 3731, pp. 275–289. Springer, Heidelberg (2005)
5. Qian, K., Nymeyer, A., Susanto, S.: Experiments with multiple abstraction heuristics in symbolic verification. In: Zucker, J.-D., Saitta, L. (eds.) SARA 2005. LNCS (LNAI), vol. 3607, pp. 290–304. Springer, Heidelberg (2005)

6. Groce, A., Visser, W.: Heuristics for model checking Java programs. Int. Journal on Software Tools for Technology Transfer (STTT) 6(4), 260–276 (2004)
7. Kupferschmid, S., Dräger, K., Hoffmann, J., Finkbeiner, B., Dierks, H., Podelski, A., Behrmann, G.: Uppaal/DMC - abstraction-based heuristics for directed model checking. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 679–682. Springer, Heidelberg (2007)
8. Qian, K., Nymeyer, A.: Guided invariant model checking based on abstraction and symbolic pattern databases. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 497–511. Springer, Heidelberg (2004)
9. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
10. Nymeyer, A., Qian, K.: Heuristic search algorithm based on symbolic data structures. In: Gedeon, T(T.) D., Fung, L.C.C. (eds.) AI 2003. LNCS (LNAI), vol. 2903, pp. 966–979. Springer, Heidelberg (2003)
11. Owen, D., Menzies, T., Heimdahl, M., Gao, J.: On the advantages of approximate vs. complete verification: Bigger models, faster, less memory, usually accurate. In: Proc. of IEEE/NASA Softw. Eng. Workshop, SEW 2003, pp. 75–81 (2003)
12. Grosu, R., Smolka, S.A.: Monte Carlo model checking. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 271–286. Springer, Heidelberg (2005)
13. Qian, K., Nymeyer, A.: Abstraction-based model checking using heuristical refinement. In: Wang, F. (ed.) ATVA 2004. LNCS, vol. 3299, pp. 165–178. Springer, Heidelberg (2004)
14. Clarke, E., Grumberg, O., Long, D.: Model checking and abstraction. ACM Transactions on Programming Languages and Systems 16(5), 1512–1542 (1994)
15. Haslum, P.: Model checking by random walk. In: Proc. of ECSEL Workshop (1999)
16. Pelánek, R., Hanžl, T., Černá, I., Brim, L.: Enhancing random walk state space exploration. In: Proc. of Formal Methods for Industrial Critical Systems, FMICS 2005, pp. 98–105. ACM Press, New York (2005)

Parallel Processes with Real-Time and Data: The ATLANTIF Intermediate Format

Jan Stöcker, Frédéric Lang, and Hubert Garavel

Vasy project-team, Inria Grenoble – Rhône-Alpes/Lig, Montbonnot, France
{Jan.Stoecker,Frederic.Lang,Hubert.Garavel}@inria.fr

Abstract. To model real-life critical systems, one needs "high-level" languages to express three important concepts: complex data structures, concurrency, and real-time. So far, the verification of timed systems has been successfully applied to "low-level" models, such as timed extensions of automata or of Petri nets. To bridge the gap between high-level languages, which allow a concise modeling of systems, and low-level models, for which efficient algorithms and tools have been designed, intermediate models are needed. In this paper, we propose the Atlantif intermediate model, an extension with real-time and concurrency of the Ntif (*New Technology Intermediate Format*) intermediate model. We define the formal semantics of Atlantif and present a translator from Atlantif to timed automata (for verification using Uppaal), and to time Petri nets (for verification using Tina).

1 Introduction

In many cases, asynchronous real-time systems can be modeled as a set of processes that run in parallel, communicate, synchronize mutually, and are subject to quantitative time constraints. The description and verification of asynchronous real-time systems has been a very active research subject, which has led to numerous theoretical results established upon various low-level models, such as timed automata [1, 11], timed extensions of Petri nets [32, 16], and timed process algebras [17, 31, 10, 18, 9, 40, 3, 34, 35, 28]. These models have been at the basis of successful verification tools, such as AltaRica [15], Kronos [41], Red [39], Romeo [26], Rtl [17], Tina [5], Uppaal [30], etc.

However, although appropriate for verification, these models are often too low-level for describing complex systems concisely. Higher-level models are thus needed. Such models should allow the expression of three aspects formally and simultaneously:

1. The first aspect is *data*, ranging from simple types (such as booleans, integers and enumerated types) to structured types (such a arrays, lists, unions, and trees). This also includes functions, either predefined or user defined.
2. The second aspect is *control*, such as communication, synchronization between processes, and the ability for processes to activate and/or deactivate each others.

3. The third aspect is *real-time*, such as *delays* (inaction of a process during a predefined time), constraints on the time instants when a process can communicate, *urgency* (indicating that a communication must not be delayed), and *latency* [17] (indicating that some time can elapse before a communication becomes urgent).

This scientific goal has been addressed since the late 80's, with the definition of high-level formal models that combine the strong theoretical foundations of process algebras with language features suitable for a wider industrial dissemination of formal methods [36, 31], converging into the E-Lotos language standardized by Iso [28]. On the other hand, several semi-formal industrial models based on model-driven tool development are emerging, such as Aadl [20], SysML [27] and Uml/Marte [19]. However, in both cases, verification tools are still lacking for this models. This could be adressed by translators from these high-level models to the low-level models accepted by existing verification tools. Suitable intermediate models are thus needed to enable a better integration of timed verification in industrial tool chains.

Related Work. Ntif (*New Technology Intermediate Form*) [22] is a minimal intermediate model for processes with sequential control and complex data. An Ntif process is an automaton that consists of a set of control states, to each of which is associated a statement called a *multibranch* transition and defined using high-level standard control structures (deterministic and nondeterministic variable assignments, **if-then-else** and **case** conditionals, nondeterministic choice, **while** loops, etc.) and communication events. This allows a representation of processes that is more compact than condition/action models such as If [14], Bip [4], and LPEs [37] and that can be easily translated into such models.

More recently, Ntif found industrial applications in the framework of the Topcased[1] project led by Airbus. The concepts of Ntif served as a basis for Fiacre (*Format Intermédiaire pour les Architectures de Composants Répartis Embarqués*) [6], an intermediate model between industrial models and verification tools. Transformations from Aadl and Sdl into Fiacre have been specified, and Fiacre has been connected to two model checkers: Cadp [24] and Tina [8].

Contribution. As a basis to design the future revisions of Fiacre, we propose in this paper an enhanced version of Ntif named Atlantif, which provides more general concurrency and real-time constructs. As regards control, Atlantif provides a mechanism to synchronize processes, based on a generalization of synchronization vectors. As regards real-time, it associates delays and time constraints to communications, following the line of prior work that led to the definition of real-time process algebras, such as ET-Lotos [31], RT-Lotos [17], and E-Lotos. Atlantif has a formal semantics that is intended to allow semantic-preserving translations from high-level languages into low-level models, and that satisfies suitable properties such as *time additivity* (every sequence of timed transitions can be collapsed into a single timed transition), *time determinism* (elapsing of a

[1] `http://www.topcased.org`

certain amount of time leads to a unique state) and *maximal progress of urgent actions* (time cannot elapse if an urgent action is possible) [33].

In order to assess our choices, we also present a prototype translator tool from ATLANTIF to lower-level models, thus enhancing the cooperation between different methods. It targets timed automata, suitable as input for the UPPAAL model checker and time Petri nets, suitable as input for the TINA model checker. We illustrate the benefits of ATLANTIF and its translators on four examples borrowed from the literature of real-time models.

Paper outline. In Section 2, we present the syntax and formal semantics of ATLANTIF. In Section 3, we show how subsets of ATLANTIF can be translated into UPPAAL's timed automata and TINA's time Petri nets, we present a tool, and we give examples. In Section 4, we give some concluding remarks.

2 Overview of ATLANTIF

2.1 Syntax

The syntax of ATLANTIF, given in Fig. 1, is described in EBNF (*Extended Backus-Naur Form*), where parts between square brackets are optional and vertical bars denote alternatives. ATLANTIF is a strict superset of NTIF; shading is used to highlight these extensions, which will be detailed in Sections 2.2 and 2.3.

For conciseness, we will not detail type definitions (including complex data types, such as records, lists, etc.), type constructors, and function definitions. There are also ATLANTIF constructs (mechanisms to start and stop processes, to share variables between processes, and to perform synchronizations that do not induce discrete transitions) that will not be detailed in this paper.

2.2 Sequential Processes in ATLANTIF

An ATLANTIF sequential process, called a *unit*, contains variable declarations and a list of discrete states, the first of which is taken to be the initial state. To each discrete state s we associate a *multibranch* transition of the form "**from** s A", where A is an action, noted $act(s)$. Contrary to usual models, in which actions are simply "condition/assignment" pairs, ATLANTIF actions are built using high-level language constructs combining atomic actions. A particular action is *gate communication*, which allows data exchange in the form of offers, each of which represents either the emission ("$!E$") of some value expression E or the reception ("$?P$") of some value that is decomposed against a pattern P using pattern-matching.

As regards real-time, ATLANTIF supports either discrete time (corresponding to a time domain isomorphic to $\mathbb{N}$) or dense time (corresponding to $\mathbb{R}_{\geq 0}$), as well as untimed behaviour. This timing option is given in the header of a specification (by the keywords "**no time**", "**discrete time**", or "**dense time"**) and taken to be "**no time**" if unspecified. ATLANTIF also has a "**wait**" action allowing a given amount of time to elapse (borrowed from process algebras such as TCSP [36]), and the following optional additions to gate communication:

$X ::=$ **module** M **is**	
[(**no** | **discrete** | **dense**) **time**]	(*timing options*)
type T_1 **is** $D_1 \dots$ **type** T_n **is** D_n	(*type declarations*)
function F_1 **is** $Y_1 \dots$ **function** F_l **is** Y_l	(*function declarations*)
$R_1 \ \dots \ R_m$	(*synchronizers, defined below*)
$U_0 \ \dots \ U_l$	(*unit definitions, defined below*)
end module	
$U ::=$ **unit** u **is**	
[**variables** $V_0 : T_0$ [:= E_0], …, $V_n : T_n$ [:= E_n]]	(*local variables*)
from s_0 A_0 … **from** s_m A_m	(*list of transitions*)
end unit	
$A ::= V_0, \dots, V_n$ **:=** $E_0, \dots, E_n$	(*deterministic assignment*)
| $V_0, \dots, V_n$ **:= any** $T_0, \dots, T_n$ [**where** E]	(*nondeterministic assignment*)
| **reset** $V_0, \dots, V_n$	(*variable reset*)
| **wait** E	(*delay*)
| $G\ O_1 \dots O_n$ [[**must** | **may**] **in** W]	(*gate communication*)
| **to** s'	(*jump to state*)
| A_1**;** A_2	(*sequential composition*)
| **if** E **then** A_1 **else** A_2 **end** [**if**]	(*conditional*)
| **case** E **is** $P_0 \rightarrow A_0$ | ... | $P_n \rightarrow A_n$ **end** [**case**]	(*deterministic choice*)
| **select** A_0 [] ... [] A_n **end** [**select**]	(*nondeterministic choice*)
| **while** E **do** A_0 **end** [**while**]	(*loop*)
| **null**	(*inaction*)

$O ::= !E$	(*value emission*)	$E ::= V$	(*variable*)
| $?P$	(*value reception*)	| $F(E_1, \dots, E_n)$	(*function*)
		| $C(E_1, \dots, E_n)$	(*constructor*)

$P ::=$ **any** T	(*anonymous variable*)	| P_0 **where** E	(*condition*)	| (P_0)
| V	(*variable*)	| $C(E_1, \dots, E_n)$	(*constructor*)	|

$W ::=$ `[`E_1`,`E_2`]` | `]`E_1`,`E_2`]` | `[`E_1`,`E_2`[` | `]`E_1`,`E_2`[`	(*bounded interval*)
| `[`E_1`, ...[` | `]`E_1`, ...[`	(*unbounded interval*)
| W_1 **or** W_2 | W_1 **and** W_2 | `(`W_0`)`	(*combined intervals*)
$R ::=$ **sync** G [: B] **is** K **end sync**	(*synchronizer declaration*)

$K ::= u$	(*single unit*)	$N ::= n$	(*natural integer*)
| K_1 **and** K_2	(*synchronization*)	| N_1 **or** N_2	(*choice*)
| K_1 **or** K_2	(*alternative*)	$B ::=$ **visible**	(*default value*)
| N **among** `(`$K_1, \dots, K_m$`)`		| **hidden**	
| `(`K_0`)`		| **urgent**	

where terminal and non terminal symbols mean the following:

A: *action*	M: *module identifier*	u : *unit identifier*
B: *visibility specifier*	N: *cardinality list*	U : *unit*
C: *constructor identifier*	O : *communication offer*	V : *variable identifier*
D: *type definition*	P : *pattern*	W: *time window*
E: *expression*	Q : *semantic modality*	X : *module (axiom)*
F: *function identifier*	R : *synchronizer*	Y : *function definition*
G: *gate identifier*	s : *state identifier*	
K: *synchronization formula*	T : *type identifier*	

Fig. 1. ATLANTIF syntax (shading indicates additions w.r.t. NTIF)

- A *time window* W that consists of intersections ("**and**") and unions ("**or**") of open or closed intervals, where "**...**" represents infinity. The communication may happen when the time elapsed since the communication action has been reached belongs to the time window. If W is unspecified, it is taken to be "$[0, ...[$". The time window thus has the role of a *life reducer*, similar to that found in different timed process algebras such as ET-LOTOS [31].
- A *modality* Q among "**must**" or "**may**", "**must**" indicating that the communication must occur before the end of the time window (which is called the *deadline*), and "**may**" indicating that time can elapse indefinitely. If unspecified, Q is taken to be "**may**". In the classification of [13], "**may**" corresponds to *weak* timed semantics, whereas "**must**" corresponds to *strong* timed semantics. Time Petri nets and FIACRE only allow strong timed semantics, whereas timed automata and most timed extensions of LOTOS allow a combination of both, which justifies our choice in ATLANTIF.

Static semantics. As regards static semantics, ATLANTIF inherits the same rules as NTIF [22], namely well-typedness, proper initialization of variables before use, and restriction of at most one communication on each possible path of a multibranch transition. We add the constraints that no "**wait**" action is allowed in any path following a communication in a multibranch transition, and that the time window of every "**must**" communication is either unbounded or right-closed.

Dynamic semantics – definitions. As regards dynamic semantics, we need the following definitions inherited from NTIF. We assume a set *Val* of *values*, written v, v', v_0, v_1, etc. We note $\mathcal{V}$ the set of variables. Partial functions on $\mathcal{V} \to \mathit{Val}$, called *stores*, are written $\rho, \rho', \rho_0, \rho_1$, etc. We note $\mathit{dom}(\rho)$ the domain of ρ. The *update* operator $\oslash$ and the *restriction* operator $\ominus$ are defined on stores as follows:

$$\begin{aligned}
&\rho \oslash \rho' \overset{\text{def}}{=} \rho'' \text{ where } \rho''(V) = \text{if } V \in \mathit{dom}(\rho') \text{ then } \rho'(V) \text{ else } \rho(V)\\
&\rho \ominus \{V_1, \ldots, V_n\} \overset{\text{def}}{=} \rho'' \text{ where } \mathit{dom}(\rho'') = \mathit{dom}(\rho) \setminus \{V_1, \ldots, V_n\}\\
&\qquad\qquad\qquad\qquad\quad \text{and } (\forall V \in \mathit{dom}(\rho''))\ \rho''(V) = \rho(V)
\end{aligned}$$

The semantics of expressions is given by a predicate $\mathit{eval}(E, \rho, v)$ that is **true** iff the evaluation of expression E in store ρ yields a value v. The semantics of patterns is given by a *pattern-matching* function $\mathit{match}(v, \rho, P)$ that returns either "**fail**" if v does not match P, or else a new store ρ' corresponding to ρ in which the variables of P have been assigned by the matching sub-terms of v. The semantics of offers is given by a function $\mathit{accept}(v, \rho, O)$, defined by:

$$\begin{aligned}
&\mathit{accept}(v, \rho, !E) \overset{\text{def}}{=} \text{if } \mathit{eval}(E, \rho, v) \text{ then } \rho \text{ else } \mathbf{fail}\\
&\mathit{accept}(v, \rho, ?P) \overset{\text{def}}{=} \mathit{match}(v, \rho, P)
\end{aligned}$$

We note $\mathcal{S}$ the set of state identifiers assumed to contain a special element δ, reserved for semantics, which represents an auxiliary discrete state that denotes the termination of an action, thus enabling the execution of subsequent actions.

The following definitions are also required. We note $\mathbb{D}$ the time domain, t, t', t_0, t_1, etc. its elements, and $\mathbb{L}_1 \overset{\text{def}}{=} \{G\ v_1 \ldots v_n \mid G \in \mathbb{G},\ v_1, \ldots, v_n \in \mathit{Val}\} \cup \{\varepsilon\}$ the set of labels, where $\mathbb{G}$ denotes the set of gates and ε represents transitions without communication actions. The binary operator "+" is partially defined on $\mathbb{L}_1 \times \mathbb{L}_1 \to \mathbb{L}_1$ by $l + \varepsilon \overset{\text{def}}{=} l$, $\varepsilon + l \overset{\text{def}}{=} l$, and is undefined if both its operands are different from ε. We note $\mathbb{U}$ the set of unit identifiers and $\mathcal{U}, \mathcal{U}', \mathcal{U}_0, \mathcal{U}_1$, etc. its subsets. The semantics of time windows is given by a predicate $\mathit{win\_eval}(W, \rho, D)$ that is **true** iff the evaluation of W in store ρ yields a set of time instants D. We also define a boolean function $\mathit{up\_lim}(Q, W, \rho, t)$ returning **true** iff $Q = \mathbf{must}$ and the set D defined by $\mathit{win\_eval}(W, \rho, D)$ has a maximum equal to t.

Dynamic semantics – sequential constructs. In Ntif, the semantics of actions was defined by a relation of the form $(A, \rho) \overset{l}{\Longrightarrow} (s, \rho')$, where A is an action, ρ, ρ' are stores, $s \in \mathcal{S}$ is a discrete state, and $l \in \mathbb{L}_1$ is a label [22]. Atlantif extends this to a relation to the form $(A, d, \rho) \overset{l}{\Longrightarrow} (s, d', \rho')$, where d, d' have the form (t, μ), with t a time value (intuitively representing the time that may elapse in the current unit until the next communication), and μ a boolean (called *blocking condition*), that is equal to **true** iff time is not allowed to elapse after t. This means that the action A in the context d and ρ evolves to the *local state* (s, d', ρ') (local states are also written $\sigma, \sigma', \sigma_0, \sigma_1$, etc.), producing a transition labeled l. These rules are detailed below, where shading indicates additions w.r.t. Ntif.

$$(\mathit{null})\ \frac{}{(\mathbf{null}, d, \rho) \overset{\varepsilon}{\Longrightarrow} (\delta, d, \rho)} \qquad (\mathit{wait})\ \frac{\mathit{eval}(E, \rho, v) \wedge t \geq v}{(\mathbf{wait}\ E, (t, \mu), \rho) \overset{\varepsilon}{\Longrightarrow} (\delta, (t - v, \mu), \rho)}$$

$$(\mathit{assign}_d)\ \frac{\mathit{eval}(E_0, \rho, v_0) \wedge \ldots \wedge \mathit{eval}(E_n, \rho, v_n)}{(V_0, \ldots, V_n := E_0, \ldots E_n, d, \rho) \overset{\varepsilon}{\Longrightarrow} (\delta, d, \rho \oslash [V_0 \mapsto v_0, \ldots, V_n \mapsto v_n])}$$

$$(\mathit{assign}_n)\ \frac{v_0 \in T_0, \ldots, v_n \in T_n \wedge \rho' = \rho \oslash [V_0 \mapsto v_0, \ldots, V_n \mapsto v_n] \wedge \mathit{eval}(E, \rho', \mathbf{true})}{(V_0, \ldots, V_n := \mathbf{any}\ T_0, \ldots, T_n\ \mathbf{where}\ E, d, \rho) \overset{\varepsilon}{\Longrightarrow} (\delta, d, \rho')}$$

$$(\mathit{reset})\ \frac{}{(\mathbf{reset}\ V_0, \ldots, V_n, d, \rho) \overset{\varepsilon}{\Longrightarrow} (\delta, d, \rho \ominus \{V_0, \ldots, V_n\})} \ (\mathit{to}) \frac{}{(\mathbf{to}\ s, d, \rho) \overset{\varepsilon}{\Longrightarrow} (s, d, \rho)}$$

$$(\mathit{comm})\ \frac{(\forall j \in 1..n)\ \mathit{accept}(v_j, \rho_j, O_j) = \rho_{j+1} \neq \mathbf{fail} \wedge \mathit{win\_eval}(W, \rho_{n+1}, D) \wedge t \in D}{(G\ O_1 \ldots O_n\ Q\ \mathbf{in}\ W, (t, \mu), \rho_1) \overset{G\ v_1 \ldots v_n}{\Longrightarrow} (\delta, (t, \mathit{up\_lim}(Q, W, \rho_{n+1}, t)), \rho_{n+1})}$$

$$(\mathit{seq}_1) \frac{(A_1, d, \rho) \overset{l_1}{\Longrightarrow} (\delta, d', \rho') \wedge (A_2, d', \rho') \overset{l_2}{\Longrightarrow} \sigma}{(A_1; A_2, d, \rho) \overset{l_1 + l_2}{\Longrightarrow} \sigma} (\mathit{seq}_2) \frac{(A_1, d, \rho) \overset{l}{\Longrightarrow} (s, d', \rho') \wedge s \neq \delta}{(A_1; A_2, d, \rho) \overset{l}{\Longrightarrow} (s, d', \rho')}$$

$$(\mathit{select})\ \frac{k \in 0..n \wedge (A_k, d, \rho) \overset{l}{\Longrightarrow} \sigma}{(\mathbf{select}\ A_0\ [] \ldots []\ A_n\ \mathbf{end}, d, \rho) \overset{l}{\Longrightarrow} \sigma}$$

$$(\mathit{case})\ \frac{\begin{array}{c}\mathit{eval}(E, \rho, v)\ \wedge\ (\forall j < k)\ \mathit{match}(v, \rho, P_j) = \mathbf{fail} \\ \wedge\ \mathit{match}(v, \rho, P_k) = \rho_k \wedge\ (A_k, d, \rho_k) \overset{l}{\Longrightarrow} \sigma\end{array}}{(\mathbf{case}\ E\ \mathbf{is}\ P_0 \to A_0 \mid \ldots \mid P_n \to A_n\ \mathbf{end}, d, \rho) \overset{l}{\Longrightarrow} \sigma}$$

$$(\mathit{while}_1)\ \frac{\mathit{eval}(E, \rho, \mathbf{true})\ \wedge\ (A; \mathbf{while}\ E\ \mathbf{do}\ A\ \mathbf{end}, d, \rho)\ \overset{l}{\Longrightarrow}\ \sigma}{(\mathbf{while}\ E\ \mathbf{do}\ A\ \mathbf{end}, d, \rho)\ \overset{l}{\Longrightarrow}\ \sigma}$$

$$(\mathit{while}_2)\ \frac{\mathit{eval}(E, \rho, \mathbf{false})}{(\mathbf{while}\ E\ \mathbf{do}\ A\ \mathbf{end}, d, \rho)\ \overset{\varepsilon}{\Longrightarrow}\ (\delta, d, \rho)}$$

$$(\varepsilon\text{-}\mathit{elim})\ \frac{(A, d, \rho)\ \overset{\varepsilon}{\Longrightarrow}\ (s, d', \rho') \wedge s \neq \delta \wedge (\mathit{act}(s), d', \rho')\ \overset{l}{\Longrightarrow}\ (s', d'', \rho'')}{(A, d, \rho)\ \overset{l}{\Longrightarrow}\ (s', d'', \rho'')}$$

Fig. 2 gives an example of a system composed of a user and a lamp. The user, modeled by the *User* unit, pushes repeatedly a button using gate *Push*. Between two pushes, the user may wait indefinitely, but must wait at least one time unit. The lamp, modeled by the *Lamp* unit, has three levels of brightness, modeled by the three discrete states *Off*, *Low*, and *Bright*. When the lamp is off (state *Off*), pushing the button switches it on with low brightness (state *Low*). If the next push happens within less than 5 time units then the lamp gets brighter (state *Bright*). If it happens after 5 time units then the lamp is switched off.

```
module Light is dense time
sync Push is User and Lamp end sync
init User, Lamp (* initially started units *)
unit User is
  from Rdy
    wait 1; Push; to Rdy
end unit
unit Lamp is
  from Off
    Push; to Low
  from Low
    select Push in [0, 5[;
          to Bright
        [] Push in [5, ... [;
          to Off
    end select
  from Bright
    Push; to Off
end unit
end module
```

Fig. 2. ATLANTIF program describing a light switch

2.3 Concurrency in ATLANTIF

In ATLANTIF, a specification contains several units synchronized with respect to *synchronizers* (Fig. 1), which are a generalization of synchronization vectors [2, 12]. A synchronizer is invoked every time a unit reaches a communication action i.e., every time it wants to propose a rendezvous to its environment. Precisely, a synchronizer has the form "**sync** G : B **is** K **end sync**", where:

– G is a gate that triggers the synchronizer.
– B is an optional tag attached to G, noted $\mathit{tag}(G)$, which may take one out of three different values: "**visible**" induces a transition labeled by G and the offers exchanged on G; "**hidden**" induces an internal transition called τ-transition; and "**urgent**" behaves like the latter, but also blocks time when a synchronization is possible. If no tag is specified, the synchronizer is visible.
– K is a formula consisting of unit identifiers and boolean operators, which denotes combinations of units that must synchronize, each such combination being called a "*synchronization set*". The set of synchronization sets attached to G, noted $\mathit{sync}(G)$, is defined as follows:

$$
\begin{aligned}
&\mathit{sync}(u) = \{\{u\}\}\\
&\mathit{sync}(K_1 \textbf{ and } K_2) = \{S_1 \cup S_2 \mid S_1 \in \mathit{sync}(K_1) \wedge S_2 \in \mathit{sync}(K_2)\}\\
&\mathit{sync}(K_1 \textbf{ or } K_2) = \mathit{sync}(K_1) \cup \mathit{sync}(K_2)\\
&\mathit{sync}(n \textbf{ among } (K_1, \ldots, K_m)) = \mathit{sync}(K'_1 \textbf{ or} \ldots \textbf{or } K'_k), \text{ where}\\
&\quad \{K'_1, \ldots, K'_k\} = \{(K_{i_1} \textbf{ and} \ldots \textbf{and } K_{i_n}) \mid 1 \leq i_1 < \ldots < i_n \leq m\}\\
&\mathit{sync}(n_1 \textbf{ or} \ldots \textbf{or } n_l \textbf{ among } (K_1, \ldots, K_m)) =\\
&\quad \mathit{sync}(n_1 \textbf{ among } (K_1, \ldots, K_m) \textbf{ or} \ldots \textbf{or } n_l \textbf{ among } (K_1, \ldots, K_m))
\end{aligned}
$$

To express concurrency, other intermediate models (such as CÆSAR networks [21] or communicating state machines [29]) combine communications of processes into Petri net-like transitions. A drawback of this approach is that the number of transitions in the resulting model can be the product of the numbers of transitions in each process. Synchronizers provide a more symbolic approach that avoids these problems, while being general enough to express the following:

- Competition between synchronizing processes can be expressed by synchronizers denoting several synchronization sets e.g., in "u_1 **and** (u_2 **or** u_3)", u_2 and u_3 compete to synchronize with u_1.
- *Multiway* synchronization can be expressed by synchronization sets containing more than two units e.g., in "u_1 **and** u_2 **and** u_3", the three units u_1, u_2 and u_3 must synchronize altogether.
- The generalized parallel composition operators of [25] can also be expressed. For instance, "**par** $G\#2, G\#3$ **in** $u_1||u_2||u_3$ **end par**", which means that either two or three processes among u_1, u_2, and u_3 synchronize on G, can be expressed by "**sync** G **is** 2 **or** 3 **among** (u_1, u_2, u_3) **end sync**".

Dynamic semantics – concurrency and real-time. Contrary to NTIF, which had no parallel semantics as it was limited to sequential processes, ATLANTIF supports a second layer of semantics for concurrency and real-time. It is given by a TLTS (*Timed Labeled Transition System*) of the form $(\mathbb{S}, \mathbb{T}, S_0)$, where:

- $\mathbb{S}$ is a set of *global states* (as opposed the *local states*) of the form (π, θ, ρ) (written S, S', S_0, S_1, etc.), where $\pi : \mathbb{U} \to \mathcal{S}$ is a function, called *state distribution*, that maps each unit to its current discrete state, $\theta : \mathbb{U} \to (\mathbb{D} \times \textbf{Bool})$ is a function, called *time distribution*, that maps each unit to its current time value and blocking condition, and ρ is a store.
- $\mathbb{T}$ is a set of transitions defined as a relation in $\mathbb{S} \times \mathbb{L}_2 \times \mathbb{S}$, where $\mathbb{L}_2 \overset{\text{def}}{=} \mathbb{L}_1 \cup \{\tau\} \cup (\mathbb{D} \setminus \{0\})$. Transitions labeled in $\mathbb{D} \setminus \{0\}$ are called *timed* transitions, whereas the other transitions are called *discrete* transitions.
- $S_0 \in \mathbb{S}$ is the initial state, which is defined by $S_0 \overset{\text{def}}{=} (\pi_0, \theta_0, \rho_0)$, where π_0 is a function that maps each unit to its initial discrete state (defined implicitly as the first discrete state in the corresponding unit), $\theta_0 : \mathbb{U} \mapsto (\mathbb{D} \times \textbf{Bool})$ is the function that constantly returns $(0, \textbf{false})$, and ρ_0 is the store that maps each variable to its initial value, if any.

We define the following predicates:

– The predicate $\mathit{enabled}(S, l, \mu, S')$, defined on $\mathbb{S} \times (\mathbb{L}_1 \setminus \{\varepsilon\}) \times \mathbf{Bool} \times \mathbb{S}$, is **true** iff (1) a transition labeled l may occur in global state S and leads to global state S' and (2) the disjunction of the blocking conditions in the local states reached via this transition equals μ. Formally:

$$\begin{array}{l} \mathit{enabled}((\pi,\theta,\rho), G\ v_1 \ldots v_n, \mu, (\pi',\theta',\rho')) \overset{\text{def}}{=} (\exists\{u_1,\ldots,u_m\} \in \mathit{sync}(G)) \\ \quad (\forall i \in 1..m)\ (\mathit{act}(\pi(u_i)), \theta(u_i), \rho) \overset{G\ \ v_1\ldots v_n}{\Longrightarrow} (s_i, (t_i, \mu_i), \rho_i) \wedge s_i \neq \delta \wedge \\ \quad \mu = \bigvee_{i=1..m} \mu_i \wedge \pi' = \pi \oslash [u_i \mapsto s_i \mid i \in 1..m] \wedge \\ \quad \theta' = \theta \oslash [u_i \mapsto (0, \mathbf{false}) \mid i \in 1..m] \wedge \rho' = \rho \oslash \rho_1 \oslash \ldots \oslash \rho_m \end{array}$$

– Time cannot elapse in a global state if an urgent communication is enabled i.e., a communication on a gate whose synchronizer is tagged urgent or a communication of the form "$G\ O_1 \ldots O_n$ **must in** W" when the deadline of W has been reached. The predicate $\mathit{relaxed}(S)$, defined on $\mathbb{S}$, is **true** iff time can elapse in S. Formally:

$$\begin{array}{l} \mathit{relaxed}(S) \overset{\text{def}}{=} (\forall\ G\ v_1 \ldots v_n, \mu, S') \\ \qquad \mathit{enabled}(S, G\ v_1 \ldots v_n, \mu, S') \Rightarrow (\neg\mu \wedge \mathit{tag}(G) \neq \mathbf{urgent}) \end{array}$$

Discrete transitions are defined by rule (rdv) as follows:

$$(\mathit{rdv})\ \frac{\mathit{enabled}((\pi,\theta,\rho), G\ v_1 \ldots v_n, \mu, (\pi',\theta',\rho'))}{(\pi,\theta,\rho) \xrightarrow{\mathit{label}(G\ v_1 \ldots v_n)} (\pi',\theta',\rho')}$$

where function *label* transforms a non-ε label of $\mathbb{L}_1$ into a discrete label of $\mathbb{L}_2$:

$$\mathit{label}(G\ v_1 \ldots v_n) \overset{\text{def}}{=} \text{ if } \mathit{tag}(G) = \mathbf{visible} \text{ then } G\ v_1 \ldots v_n \text{ else } \tau$$

Timed transitions are defined by rule (time), which allows t units of time to elapse as long as no urgent communication is enabled. The new state is calculated by increasing all relative times by t, using "+" defined by $(\forall u)\ (\theta + t)(u) \overset{\text{def}}{=} (t_u + t, \mu_u)$ where $\theta(u) = (t_u, \mu_u)$.

$$(\mathit{time})\ \frac{t > 0 \wedge (\forall\ t' < t)\ \mathit{relaxed}((\pi, \theta + t', \rho))}{(\pi,\theta,\rho) \xrightarrow{t} (\pi, \theta + t, \rho)}$$

We illustrate the semantics by deriving two TLTS transitions for the light switch example shown in Fig. 2, page 94. We show that when *User* is in state *Rdy* and *Lamp* in state *Low*, 3 time units may elapse before the button is pushed. Formally: $(\pi,\theta,\varnothing) \xrightarrow{3} (\pi, \theta+3, \varnothing) \xrightarrow{\mathit{Push}} (\pi, \theta \oslash [\mathit{Lamp} \mapsto \mathit{Bright}], \varnothing)$, where $\pi \overset{\text{def}}{=} [\mathit{User} \mapsto \mathit{Rdy}, \mathit{Lamp} \mapsto \mathit{Low}]$, and $\theta \overset{\text{def}}{=} [\mathit{User} \mapsto (0, \mathbf{f}), \mathit{Lamp} \mapsto (0, \mathbf{f})]$ (where **f** is a shorthand for **false**).

First, $(\pi,\theta,\varnothing) \xrightarrow{3} (\pi, \theta+3, \varnothing)$ comes from the following derivation:

$$\frac{3 > 0 \wedge (\forall t' < 3) \mathit{relaxed}((\pi, \theta + t', \varnothing))}{(\pi,\theta,\varnothing) \xrightarrow{3} (\pi, \theta + 3, \varnothing)} (\mathit{time})$$

Second, $(\pi, \theta + 3, \varnothing) \xrightarrow{Push} (\pi, \theta \oslash [Lamp \mapsto Bright], \varnothing)$ comes from:

$$\frac{\{User, Lamp\} \in sync(Push) \wedge (act(Rdy), (3, \mathbf{f}), \varnothing) \overset{Push}{\Longrightarrow} (Rdy, (2, \mathbf{f}), \varnothing) \wedge \\ (act(Low), (3, \mathbf{f}), \varnothing) \overset{Push}{\Longrightarrow} (Bright, (3, \mathbf{f}), \varnothing)}{(\pi, \theta + 3, \varnothing) \xrightarrow{Push} (\pi \oslash [Lamp \mapsto Bright], \theta, \varnothing)} (rdv)$$

The premiss $(act(Rdy), (3, \mathbf{f}), \varnothing) \overset{Push}{\Longrightarrow} (Rdy, (2, \mathbf{f}), \varnothing)$ comes from the following, recalling that $act(Rdy) =$ "**wait** 1; *Push*; **to** *Rdy*":

$$\frac{\dfrac{eval(1, \varnothing, \mathbf{f}) \wedge 3 \geq 1}{(\mathbf{wait}\ 1, (3, \mathbf{f}), \varnothing) \overset{\varepsilon}{\Longrightarrow} (\delta, (2, \mathbf{f}), \varnothing)} (wait) \quad (Push;\ \mathbf{to}\ Rdy, (2, \mathbf{f}), \varnothing) \overset{Push}{\Longrightarrow} (Rdy, (2, \mathbf{f}), \varnothing)}{(act(Rdy), (3, \mathbf{f}), \varnothing) \overset{Push}{\Longrightarrow} (Rdy, (2, \mathbf{f}), \varnothing)} (seq_1)$$

At last, the premiss $(Push;\ \mathbf{to}\ Rdy, (2, \mathbf{f}), \varnothing) \overset{Push}{\Longrightarrow} (Rdy, (2, \mathbf{f}), \varnothing)$ comes from:

$$\frac{\dfrac{}{(Push, (2, \mathbf{f}), \varnothing) \overset{Push}{\Longrightarrow} (\delta, (2, \mathbf{f}), \varnothing)} (comm) \quad \dfrac{}{(\mathbf{to}\ Rdy, (2, \mathbf{f}), \varnothing) \overset{\varepsilon}{\Longrightarrow} (Rdy, (2, \mathbf{f}), \varnothing)} (to)}{(Push;\ \mathbf{to}\ Rdy, (2, \mathbf{f}), \varnothing) \overset{Push}{\Longrightarrow} (Rdy, (2, \mathbf{f}), \varnothing)} (seq_1)$$

The premiss $(act(Low), (3, \mathbf{f}), \varnothing) \overset{Push}{\Longrightarrow} (Bright, (3, \mathbf{f}), \varnothing)$ is derived similarly by the rules $(comm)$, (to), (seq_1), and $(select)$.

With this semantic approach, we respect the standard property that time must elapse at the same speed in all units. Furthermore, the following proposition shows that this semantics has the suitable properties mentioned in Section 1.

Proposition. The TLTS corresponding to the semantics of an ATLANTIF specification satisfies the properties of (i) time additivity (two successive delays are equal to their sum), (ii) time determinism (no state allows two different successors after the same delay) and (iii) maximal progress of urgent actions (no delay is possible in states where an urgent action is possible).

Proof.

(i) Let S, S' be global states. We must show that $\forall\ t_1, t_2 \in (\mathbb{D} \setminus \{0\})$:

$$S \xrightarrow{t_1 + t_2} S' \text{ iff } (\exists\ S'')\ S \xrightarrow{t_1} S'' \text{ and } S'' \xrightarrow{t_2} S'$$

We define $S \overset{\text{def}}{=} (\pi, \theta, \rho)$. We note that time can only elapse using the $(time)$ rule, which does not modify π and ρ and increases θ by some delay. Therefore, the above statement can be rephrased as:

$$\begin{aligned} &(\pi, \theta, \rho) \xrightarrow{t_1 + t_2} (\pi, \theta + (t_1 + t_2), \rho) \\ \text{iff } &(\pi, \theta, \rho) \xrightarrow{t_1} (\pi, \theta + t_1, \rho) \text{ and } (\pi, \theta + t_1, \rho) \xrightarrow{t_2} (\pi, (\theta + t_1) + t_2, \rho) \end{aligned}$$

Given the definition of $+$, it is obvious that $\theta + (t_1 + t_2) = (\theta + t_1) + t_2$. From the premiss of rule $(time)$, we can reduce the above goal to the obvious following statement:

$$\begin{aligned} &(\forall\ t' < t_1 + t_2)\ relaxed((\pi, \theta + t', \rho)) \\ \text{iff } &(\forall\ t' < t_1)\ relaxed((\pi, \theta + t', \rho)) \text{ and } (\forall\ t' < t_2)\ relaxed((\pi, \theta + (t_1 + t'), \rho)) \end{aligned}$$

(ii) Again, we note that time can only elapse using rule $(time)$, which for given global state S and time t defines a unique successor state.

(iii) Let S be a global state allowing an urgent action, i.e. $\neg relaxed(S)$. Then the premiss of rule (*time*) cannot be satisfied in S i.e., time cannot elapse in S. $\square$

3 Automated Translations to Verification Tools

We developed a prototype translator tool, which maps Atlantif models to either the TA (*timed automata*) used by the tool Uppaal [30] or the TPN (*time Petri nets*) used by Tina [8]. Outlines of these mappings are given in this section. We assume the reader is familiar with Uppaal's TA and Tina's TPN.

Common restrictions. Some concepts of Atlantif cannot be mapped to neither Uppaal's TA nor Tina's TPN. Concretely, Atlantif models must use dense time; expressions in **wait** actions and time windows must be integer constants; nondeterministic assignments are not supported; patterns must be made up of either variables or constants exclusively. In addition, **while** loops are not yet supported in the translation to TA, although the translation would be feasible.

Translation to UPPAAL. Each Atlantif unit is mapped to a TA. Each discrete state s is mapped to a TA location (also named s) and an invariant is synthesized from the **must** constraints of multibranch transitions originating from s. The action $act(s)$ is decomposed into one TA transition for each branch of control. If a gate communication admits several synchronization sets containing the current unit, then it is split into one transition for each such synchronization set. Since TA do not allow communication offers, data exchanges are emulated using TA shared variables.

A key issue is that Uppaal's TA synchronizations involve at most two automata[2], whereas Atlantif allows multiway synchronizations involving $n > 2$ units. The solution requires that exactly one unit sends data (i.e., all offers are emissions), whereas the $(n-1)$ other units receive data (i.e., all offers are receptions): the gate communication in the sender unit is split into a sequence of $(n-1)$ communications, each of which synchronizes with a receiver.

Translation to TINA. Each Atlantif unit is mapped to a TPN. Each discrete state s is mapped to a TPN place (also named s) and the corresponding action $act(s)$ is decomposed into several TPN transitions, each TPN transition being labeled by a gate. As regards time constraints, we only consider time intervals and we implement a solution inspired from [7], that requires additional auxiliary places and transitions. Given a communication on a gate G, which corresponds to a Petri net transition T, we calculate the sum m of all delays that occur in "**wait**" actions preceding the communication. We remove these wait actions and we increase the bounds of the time window by m. The resulting time window is then implemented in the form of zero, one, or two new transitions as follows:

– If the lower bound of the time window is $n > 0$, then we add an unlabeled transition with time constraint "$[n, \omega[$" (or "$]n, \omega[$", if the bound is strict), no

[2] Uppaal also allows a broadcast communication, which is inapt for our purpose, because Uppaal's broadcast is not blocking.

out-place and a new in-place s_1. We add s_1 both to the inhibitor places of T, and to the out-places of every transition for which s is already an out-place.
- If the modality of the communication is **may** and the time window has an upper bound n, then we add an unlabeled transition with time constraint "$]n, \omega[$" (or "$[n, \omega[$", if the bound is strict), no out-place and a new in-place s_2. We add s_2 to the in-places of T and the new transition is given priority over T.
- If the modality of the communication is **must** and the time window has an upper bound n, then we add an unlabeled transition with time constraint "$[n, n]$", no out-place and a new in-place s_3. We add s_3 to the in-places of T and all transitions except those created for other **must** constraints are given priority over this new unlabeled transition.

The TPNs corresponding to each unit are combined into a single one by merging synchronizing transitions, using the method described in [7].

Tool implementation. Our prototype translator was implemented using the method proposed in [23] and consists of 538 lines of C code, 2,193 lines of SYNTAX code, and 13,146 lines of LOTOS NT code. The tool architecture is schematized in Fig. 3.

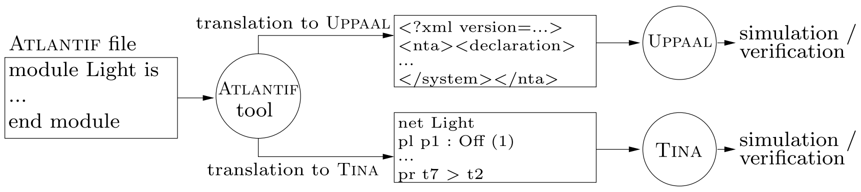


Fig. 3. The ATLANTIF to UPPAAL / TINA translation tool

We applied this translator to four examples, namely the light switch presented in Fig. 2 (page 94), the CSMA/CD protocol, which is a common benchmark specification [41], a stop-and-wait protocol, implemented with one sender, one receiver and two transmission channels, and a train gate controller. The translations into TA and TPN of the light switch example are shown in Fig. 4 and 5 respectively.

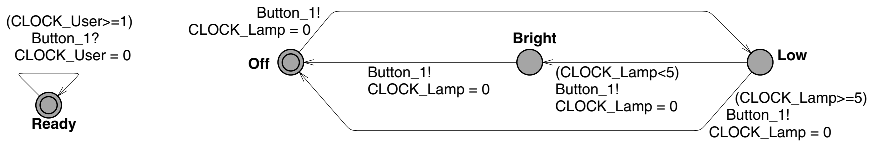


Fig. 4. The two automatically generated UPPAAL TA for the light switch example

Fig. 6 compares the size of ATLANTIF programs with the size of the corresponding TA and TPN. It shows that ATLANTIF enables shorter descriptions, in particular due to its concise syntax for time and its ability to define multiway

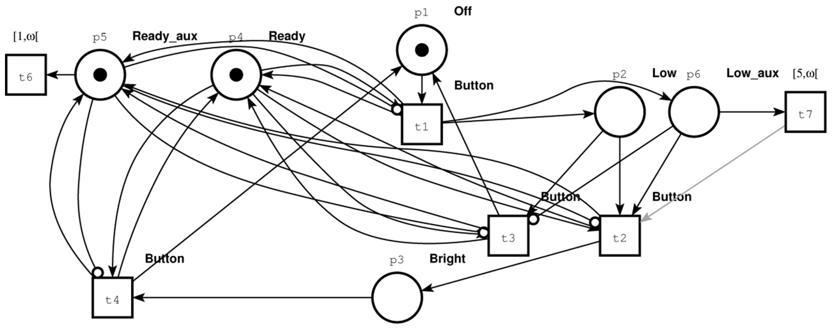


Fig. 5. The automatically generated Tina time Petri net for the light switch example

	Atlantif		Uppaal-TA		Tina-TPN	
	disc. states	trans.	locations	trans.	places	trans.
Light switch	4	4	4	5	6	6
CSMA/CD (3 Stations)	12	12	14	42	40	142
Stop-and-wait	10	10	10	12	29	56
Train Gate Controller	12	12	18	18	23	18

Fig. 6. Size comparison: Atlantif vs. generated Uppaal vs. generated Tina

synchronizations. Note that the number of locations of the TA generated for the CSMA/CD is the same as in a handwritten specification available on the web[3].

These results suggest that the TA translation is efficient for programs with multiple occurrences of simple synchronizers (i.e., synchronizers involving at most two units), whereas the TPN translation is efficient for limited occurrences of more complex synchronizers.

4 Conclusion

This paper proposes Atlantif, a simple and elegant extension of the intermediate model Ntif [22] with concurrency and real-time, intended for a better integration of formal verification tools in industrial environments. Thus, Atlantif supports the three main concepts needed to model complex asynchronous real-time systems: elaborate data types, concurrency, and quantitative time.

Atlantif has a simple timed semantics, where time elapsing is concentrated in a single rule, which satisfies time additivity, time determinism, and maximal progress. This goal is not obvious to achieve: for example, complex syntactic restrictions had to be brought to E-Lotos to ensure those properties; as another example, RT-Lotos does not satisfy time additivity.

We also presented a translator mapping Atlantif to two advanced verification tools, Uppaal [30] and Tina [8].

As regards future work, we plan to extend our translator with new features and to use it on larger industrial examples. Atlantif could also be a basis to enhance the Fiacre intermediate model [6] used in the Topcased project.

[3] http://www.it.uu.se/research/group/darts/uppaal/benchmarks/#CSMA

References

[1] Alur, R., Dill, D.L.: A Theory of Timed Automata. Theoretical Computer Science 126(2), 183–235 (1994)

[2] Arnold, A.: MEC: A System for Constructing and Analysing Transition Systems. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407. Springer, Heidelberg (1990)

[3] Baeten, J., Middelburg, C.: Real time and discrete time. In: Process Algebra with Timing. North-Holland, Amsterdam (2001)

[4] Basu, A., Bozga, M., Sifakis, J.: Modeling Heterogeneous Real-time Components in BIP. In: Proc. of SEFM. IEEE Computer Society Press, Los Alamitos (2006)

[5] Berthomieu, B., Diaz, M.: Modeling and Verification of Time Dependent Systems Using Time Petri Nets. IEEE Transactions on Software Engineering 17(3), 259–273 (1991)

[6] Berthomieu, B., Garavel, H., Lang, F., Vernadat, F.: Verifying Dynamic Properties of Industrial Critical Systems Using TOPCASED/FIACRE. ERCIM News 75, 32–33 (2008)

[7] Berthomieu, B., Peres, F., Vernadat, F.: Bridging the Gap Between Timed Automata and Bounded Time Petri Nets. In: Asarin, E., Bouyer, P. (eds.) FORMATS 2006. LNCS, vol. 4202, pp. 82–97. Springer, Heidelberg (2006)

[8] Berthomieu, B., Vernadat, F.: Time Petri Nets Analysis with TINA. In: Proc. of QEST (2006)

[9] Blom, S., Ioustinova, N., Sidorova, N.: Timed Verification with μCR. In: Broy, M., Zamulin, A.V. (eds.) PSI 2003. LNCS, vol. 2890, pp. 178–192. Springer, Heidelberg (2004)

[10] Bolognesi, T., Lucidi, F.: LOTOS-like Process Algebras with Urgent or Timed Interactions. In: Proc. of FORTE 1991. North-Holland, Amsterdam (1991)

[11] Bornot, S., Sifakis, J., Tripakis, S.: Modeling urgency in timed systems. In: de Roever, W.-P., Langmaack, H., Pnueli, A. (eds.) COMPOS 1997. LNCS, vol. 1536, p. 103. Springer, Heidelberg (1998)

[12] Bouali, A., Ressouche, A., Roy, V., de Simone, R.: The Fc2Tools set: a Toolset for the Verification of Concurrent Systems. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102. Springer, Heidelberg (1996)

[13] Boyer, M., Roux, O.H.: Comparison of the Expressiveness of Arc, Place and Transition Time Petri Nets. In: Kleijn, J., Yakovlev, A. (eds.) ICATPN 2007. LNCS, vol. 4546, pp. 63–82. Springer, Heidelberg (2007)

[14] Bozga, M., Graf, S., Ober, I., Ober, I., Sifakis, J.: Tools and Applications II: The IF Toolset. In: Proc. of SFM (2004)

[15] Cassez, F., Pagetti, C., Roux, O.: A timed extension for AltaRica. Fundamenta Informaticæ 62(3-4), 291–332 (2004)

[16] Cerone, A., Maggiolo-Schettini, A.: Time-based expressivity of Time Petri Nets for system specification. Theoretical Computer Science 216(1), 1–54 (1999)

[17] Courtiat, J.-P., Cruz de Oliveira, R.: On RT-LOTOS and its Application to the Formal Design of Multimedia Protocols. Annals of Telecommunications 50(11-12), 888–906 (1995)

[18] Davies, J.W., Schneider, S.A.: A Brief History of Timed CSP. Theoretical Computer Science 138(2), 243–271 (1995)

[19] Faugère, M., Bourbeau, T., de Simone, R., Gérard, S.: MARTE: Also an UML Profile for Modeling AADL Applications. In: Proc. of ICECCS. IEEE, Los Alamitos (2007)

[20] Feiler, P., Gluch, D., Hudak, J.: The Architecture Analysis & Design Language (AADL): An Introduction. Technical note, Carnegie Mellon (2006)

[21] Garavel, H.: Compilation et vérification de programmes LOTOS. PhD thesis, Université Joseph Fourier, Grenoble (1989)

[22] Garavel, H., Lang, F.: NTIF: A General Symbolic Model for Communicating Sequential Processes with Data. In: Peled, D.A., Vardi, M.Y. (eds.) FORTE 2002. LNCS, vol. 2529. Springer, Heidelberg (2002); Full version available as INRIA Research Report RR-4666
[23] Garavel, H., Lang, F., Mateescu, R.: Compiler Construction Using LOTOS NT. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, p. 9. Springer, Heidelberg (2002)
[24] Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In: Damm, W., Hermanns, H. (eds.) CAV 2007. H. Garavel, F. Lang, R. Mateescu, and W. Serwe, vol. 4590, pp. 158–163. Springer, Heidelberg (2007)
[25] Garavel, H., Sighireanu, M.: A Graphical Parallel Composition Operator for Process Algebras. In: Proc. of FORTE/PSTV. Kluwer, Dordrecht (1999)
[26] Gardey, G., Lime, D., Magnin, M., Roux, O.: Romeo: A Tool for Analyzing Time Petri Nets. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 418–423. Springer, Heidelberg (2005)
[27] Hause, M.: The SysML Modelling Language. In: Fifteenth European Systems Engineering Conference (2006)
[28] ISO/IEC. Enhancements to LOTOS (E-LOTOS). International Standard 15437:2001, International Organization for Standardization (September 2001)
[29] Karjoth, G.: Implementing LOTOS Specifications by Communicating State Machines. In: Cleaveland, W.R. (ed.) CONCUR 1992. LNCS, vol. 630. Springer, Heidelberg (1992)
[30] Larsen, K., Pettersson, P., Yi, W.: UPPAAL in a Nutshell. Int. Journal on Software Tools for Technology Transfer 1(1-2), 134–152 (1997)
[31] Léonard, L., Leduc, G.: A Formal Definition of Time in LOTOS. In: Formal Aspects of Computing, pp. 28–96 (1998)
[32] Merlin, P.M.: A study of the recoverability of computing systems. PhD thesis, Univ. of California, Irvine (1974)
[33] Nicollin, X., Sifakis, J.: An Overview and Synthesis on Timed Process Algebras. In: de Bakker, J.W., de Roever, W.-P., Rozenberg, G. (eds.) REX 1992. LNCS, vol. 666. Springer, Heidelberg (1993)
[34] Nicollin, X., Sifakis, J.: The Algebra of Timed Processes ATP: Theory and Application. Information and Computation 114(1), 131–178 (1994)
[35] Ouaknine, J., Worrell, J.: Timed CSP = closed timed ε-automata. Nordic Journal of Computing 10(2), 99–133 (2003)
[36] Reed, G.M., Roscoe, A.W.: A Timed Model for Communicating Sequential Processes. Theoretical Computer Science 58, 249–261 (1988)
[37] Reniers, M.A., Usenko, Y.S.: Analysis of Timed Processes with Data Using Algebraic Transformations. In: Proc. of TIME. IEEE, Los Alamitos (2005)
[38] Sadani, T., Boyer, M., de Saqui-Sannes, P., Courtiat, J.-P.: Effective representation of RT-LOTOS terms by finite time petri nets. In: Najm, E., Pradat-Peyre, J.-F., Donzeau-Gouge, V.V. (eds.) FORTE 2006. LNCS, vol. 4229, pp. 404–419. Springer, Heidelberg (2006)
[39] Wang, F.: Symbolic Simulation-Checking of Dense-Time Automata. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 352–368. Springer, Heidelberg (2007)
[40] Yi, W.: CCS + Time = An Interleaving Model for Real Time Systems. In: Leach Albert, J., Monien, B., Rodríguez-Artalejo, M. (eds.) ICALP 1991. LNCS, vol. 510. Springer, Heidelberg (1991)
[41] Yovine, S.: Kronos: A verification tool for real-time systems. International Journal of Software Tools for Technology Transfer 1(1/2), 123–133 (1997)

Changing System Interfaces Consistently: A New Refinement Strategy for CSP||B

Steve Schneider and Helen Treharne

Department of Computing, University of Surrey

Abstract. This paper introduces action refinement in the context of CSP||B. Our motivation to include this notion of refinement within the CSP||B framework is the desire to increase flexibility in the refinement process. We introduce the ability to change the events of a CSP process and the B machines when refining a system. Notions of refinement based on traces and on traces/divergences are introduced in which abstract events are refined by sequences of concrete events. A complementary notion of refinement between B machines is also introduced, yielding compositionality results for refinement of CSP||B controlled components. The paper also introduces a notion of I/O refinement into our action refinement framework.

1 Introduction

This paper introduces an approach to event refinement in the context of CSP||B. Event refinement (or action refinement) is concerned with developing a finer level of granularity in specifications, by expanding atomic events within the description into more detailed structures. One motivation for our attention to this issue within the CSP||B framework [14] is the desire to increase our range of options when refining processes and operations. We have recently found it useful in the setting of an industrial CSP||B case study [13] to change the level of granularity of the description during the refinement process.

The challenge of how best to do this has been an issue within process algebra since at least the late 1980's, and a broad survey of the work can be found in [4, Chapter 16]. However, the integration of data refinement with action refinement has received limited attention to date. An early paper in this area is [7], which takes a state-based (Z) approach to refining atomic operations by sequences of operations. In this paper we aim to provide a framework for this notion of refinement in the context of the CSP||B combined formal method, using the CSP aspect to capture the action refinements in a more natural way. We introduce the ability to change the events of a CSP process and hence the B machines during a refinement of a system. An important feature of the new refinement framework is that it does not compromise the existing CSP||B theory and does not change the notations of CSP or classical B.

The CSP||B approach favours separation between behavioural patterns and state descriptions. However, behavioural patterns and state may need to be changed during a refinement. For example, a communication protocol may receive a message and subsequently perform some computation. At an abstract level it would be appropriate to denote the type of the message as a deferred set but in a refinement the message may be split into several smaller, more detailed, messages of a concrete type. Furthermore, the subsequent computation could also be segmented. The paper explores what it means to split events in a refinement, and whether the inputs and outputs of operations (and their types) can be changed in a refinement, or distributed across several operations.

The main contribution of the paper is a framework for event refinement: a collection of definitions of how such a notion of refinement may be naturally expressed, together with some theorems that establish that these definitions are collectively consistent. These culminate in Theorem 4, where we see the conditions given in the various definitions support a compositionality result: that refinement of components separately ensures refinement of their parallel combination. Conjecture 1 gives the corresponding result for operations with input and output.

2 CSP||B Overview

A CSP *controlled component* consists of a CSP process P in parallel with a B machine M.

CSP controllers. Controllers will be written in a subset of the CSP process algebraic language [9,11]. We begin with the following simple controller language:

Definition 1 (Controller Syntax)

$$P ::= a \rightarrow P \mid P_1 \Box P_2 \mid STOP \mid S$$

The event a is drawn from the set of events, and S is a CSP process variable. Events can either be pure CSP events, or correspond to operations in the controlled B machine. Notationally we will use e for simple atomic CSP events not corresponding to operations, whereas a will be used for operation names. S is a process variable. Recursive definitions are then given as $S \mathrel{\widehat{=}} P$. In a controller definition, all process variables used are bound by some recursive definition.

More generally, events can consist of channels communicating values. An event will then have the structure $c.v$, where c is the channel name and v is the value being passed on the channel. In general, channels can carry multiple values. The process $c!v?x \rightarrow P(x)$ denotes a process ready to output v on channel c, and to input a value x at the same time. Its subsequent behaviour is described by $P(x)$.

B machines. The B-method [1] is structured around B-machines, which provide an encapsulation of state (which can be abstract mathematical structures) and operations on that state, in an object-style structure. A machine is introduced with a name, state variables, an invariant (including type information) on those variables, an initialisation, and a collection of operations on the state.

Operations are declared as $out \longleftarrow op(in) \mathrel{\widehat{=}} PRE\ P\ THEN\ S\ END$, where P is the precondition of the operation, and S is its body. in and out can in general be sequences of formal parameters. S is an abstract assignment describing how the state can be updated. This can include single and concurrent updates, and nondeterministic choice. Initialisation is also given as an abstract assignment. The abstract assignment constructions we use in this paper are assignment: $x := E$; precondition: $PRE\ P\ THEN\ S\ END$ which executes S if P is true, but otherwise its behaviour is undetermined; parallel assignment: $S \parallel T$; and sequential composition $S; T$.

A machine is *consistent* if its invariant I is initially true, and is preserved by all of the machine's operations when called within their preconditions. The B-Method uses weakest precondition semantics to establish that machines are consistent, and we will assume machine consistency for the purposes of this paper (i.e. the results apply only for consistent machines). The notation $[S]I$ denotes the weakest precondition required for statement S to guarantee achieving post-condition I. Invoking a preconditioned operation cannot guarantee anything (not even termination) if the precondition is false, thus $[PRE\ P\ THEN\ S\ END]I = P \wedge [S]I$: to guarantee establishing I, P must initially be true, and furthermore S must establish I.

Refinement may be considered between two machines M and M'. A linking invariant J is a predicate on the states of both M and M' that is used to capture the relationship between their states, to identify when an abstract state is matched by a concrete state. The proof obligation $I \wedge J \Rightarrow [T](\neg[S]\neg J)$ is used to establish that the concrete statement T is a refinement of the abstract statement S in such a context. Further explanation can be found in [1,12].

Controlled components. A component is a controller definition P and an associated B machine M. The operations a in the machine correspond to events of the same name a in the controller. Operations $out_a \longleftarrow a(in_a)$ are matched by complementary channel communications $a?out_a!in_a$ in the controller: input in_a to the machine is provided by (i.e. an output from) the controller; and output out_a is read by (i.e. input to) the controller. The alphabet αM of the machine is given by its set of operations. We require that $\alpha M \subseteq \alpha P$, that every operation also occurs in the controller. However, controllers may also use CSP events not included in the machine, for interacting with other parts of a larger system, or with its environment.

Morgan's CSP semantics for action systems [10] allows traces, failures, and divergences to be defined for B machines in terms of the sequences of operations that they can and cannot engage in. This gives a way of considering B machines as CSP processes, and treating them within the CSP framework. This enables us to give $P \parallel M$ a CSP semantics.

The traces of a machine M are those sequences of operations $tr = \langle a_1, \ldots, a_n \rangle$ which are possible for the machine. In weakest precondition semantics, an *impossible* trace tr is miraculous: it establishes *false*, i.e. $[T; tr]false$ (where T is the initialisation of the machine). Hence the negation characterises the traces of the machine: $\neg[T;tr]false$. Thus $traces(M) = \{tr \mid \neg[T;tr]false\}$.

A sequence of operations tr is a *divergence* if the sequence of operations is not guaranteed to terminate, i.e. $\neg[T;\ tr]true$. Thus $divergences(M) = \{tr \mid \neg[T;tr]true$

These two definitions provide the link between the weakest precondition semantics of the operations, and the CSP semantics of the B machine. This definition means that calling an operation outside its precondition yields a divergence: termination cannot be guaranteed.

3 The Basic Refinement Framework without i/o

To develop the basic framework we will begin by considering pure operations and events, without any input or output communication on them. This will enable us to focus on the sequences of events that we wish to consider. Input/Output considerations will be introduced later, in Section 4.

3.1 Sequence Notation

We use the following notation in the paper. If A is a set, then A^* is the set of finite sequences of elements of A, and A^+ denotes the non-empty finite sequences of elements of A. The empty sequence is denoted $\langle\rangle$, and the concatenation of sequences s and t is denoted $s \frown t$. We write $s \leqslant t$ to denote that s is a prefix of t. If A is a set, then $s \upharpoonright A$ is the maximal subsequence of s all of whose elements are in A: projection of s to A. We also define the downwards and upwards closure on a set of sequences S respectively as follows:

$$\downarrow S \mathrel{\widehat{=}} \{tr \mid \exists\, tr' \in S.tr \leqslant tr'\} \qquad\qquad \uparrow_A S \mathrel{\widehat{=}} \{tr \in A^* \mid \exists\, tr' \in S.tr' \leqslant tr\}$$

If the set A is implicit from the context then we may write $\uparrow S$.

3.2 Implementation Mappings

We can now give a definition of consistent refinement between two consistent components $P \parallel M$ and $P' \parallel M'$. The key underlying idea is that whenever an event in an abstract controller P is substituted by a sequence of concrete events in a concrete controller's execution P', and the new concrete events correspond to B operations in a machine M', then we can guarantee that the concrete controlled component is a consistent refinement of the abstract one. We shall see that care will need to be taken when we re-use operations from M in the concrete component.

To do this, we must first introduce an implementation mapping *imp* as follows, which will need to be instantiated for each proposed component refinement.

Definition 2 (implementation mapping for events). *An* implementation mapping *is a function $imp \in A \rightarrow C^+$, from abstract events to a sequence of concrete events.*

Here $A = \alpha P$ and $C = \alpha P'$. Note that A and C do not have to be disjoint, therefore we must take into account what happens when events do not change in a refinement. We require a healthiness condition $\mathbf{IMP}_1$ on *imp* as follows:

$$\forall\, b \in A \cap C \;.\; imp(b) = \langle b \rangle$$

We also require that M and M' have the same intersection with $A \cap C$, and that their definitions for those operations in the intersection are identical. Since we are aiming for refinement, this requirement states that these elements do not change in the refinement step.

Observe that implementation mappings are different to CSP *alphabet renamings*, which map events to single events rather than to sequences.

We now define a mapping from sequences of abstract events to sequences of concrete traces.

Definition 3 (implementation mapping). *Given an implementation mapping imp, the function $\phi_{imp} : A^* \to C^*$ is defined as follows:*

$$\phi_{imp}(\langle\rangle) = \langle\rangle \qquad\qquad \phi_{imp}(\langle a \rangle \frown tr) = imp(a) \frown \phi(tr)$$

If the mapping imp is clear from the context, then it may be elided and we write $\phi(tr)$. Note that in functional terms, $\phi_{imp} = \mathit{flatten} \circ (\mathit{map}\ imp)$.

When we come to consider divergences we will need one further construction: the set of non-empty prefixes of concrete traces:

Definition 4. *Given an implementation mapping $imp : A \to C^+$ and $a \in A$, we define $imp^+(a) \mathrel{\widehat=} \{tr' \mid tr' \le imp(a) \wedge tr' \neq \langle\rangle\}$.*

This is used in the following definition: the mapping ψ identifies all subsequences related to an abstract sequence of events including the complete subsequences. This definition will be used in Theorem 4 to track down the point at which a concrete trace diverges.

Definition 5 (subsequence implementation mapping). *The function $\psi : A^* \to C^*$ is defined as follows:*

$$\psi(\langle\rangle) = \langle\rangle \qquad\qquad \psi(tr \frown \langle a \rangle) = \{\phi(tr)\} \frown imp^+(a)$$

3.3 Refinement

Having identified correspondences between abstract and concrete traces, through the function ϕ_{imp}, we are now in a position to define a corresponding notion of refinement:

Definition 6 (trace refinement relative to *imp*)

$P \sqsubseteq^T_{imp} P'$ iff $traces(P') \subseteq\ \downarrow \{\phi_{imp}(tr) \mid tr \in traces(P)\}$.

Note that refinement with respect to an implementation mapping is not preserved by parallel composition, as the following example illustrates:

Example 1. Consider $imp(a) = imp(b) = \langle c \rangle$, and

$$P = a \rightarrow STOP \qquad Q = b \rightarrow STOP \qquad P' = Q' = c \rightarrow STOP$$

Observe that $P \sqsubseteq^{T}_{imp} P'$ and $Q \sqsubseteq^{T}_{imp} Q'$ but $P \parallel Q = STOP, P' \parallel Q' = c \rightarrow STOP$, and so the refinement relation does not hold between $P \parallel Q$ and $P' \parallel Q'$.

We now obtain the following result which allows refinement of a controlled component to be deduced from the appropriate refinement relation between controllers.

Theorem 1. *If* $P \sqsubseteq^{T}_{imp} P'$ *then* $P \parallel M \sqsubseteq^{T}_{imp} P' \parallel M'$

Proof. $traces(P \parallel M) = traces(P)$ and $traces(P' \parallel M') = traces(P')$ in this case (since there is no i/o).

Observe that the machines M and M' can be independent: the result follows purely from the relationship between P and P'.

The mapping *imp* can also be used to transform a CSP process description to another CSP process which is a refinement.

Definition 7 (mapping abstract to concrete processes). *If* $imp : A \rightarrow C^{+}$ *is an implementation mapping, then we define the mapping* Θ_{imp} *on CSP process descriptions as follows:*

$$\begin{aligned}
\Theta_{imp}(STOP) &= STOP \\
\Theta_{imp}(a \rightarrow P) &= Pref(imp(a), \Theta_{imp}(P)) \\
\Theta_{imp}(P_1 \mathbin{\Box} P_2) &= \Theta_{imp}(P_1) \mathbin{\Box} \Theta_{imp}(P_2) \\
\Theta_{imp}(S) &= S
\end{aligned}$$

$$\begin{aligned}
\textit{where} \quad Pref(\langle\rangle, Q) &= Q \\
Pref((\langle b \rangle \frown tr, Q) &= b \rightarrow Pref(tr, Q)
\end{aligned}$$

The mapping has been constructed to yield the following theorem, proven by structural induction on P: that the result of the transformation is a refinement of the original process.

Theorem 2. $\forall\, imp, P \,.\, P \sqsubseteq^{T}_{imp} \Theta_{imp}(P)$

3.4 Refining B Machines

Now we consider what it means to refine a B machine in the context of an implementation mapping *imp*. This will enable the introduction of new operations during the refinement process.

Definition 8. *For a machine M and a sequence of events tr, we define $op_M(tr) = tr \upharpoonright \alpha M$. In other words, $op_M(tr)$ is the sequence of operations in tr that the machine M participates in. If the machine M is clear from the context then we may write $op(tr)$.*

Definition 9 (refinement of B machines). *If M and M' have linking invariant J, then*

$$M \sqsubseteq^B_{imp} M' \text{ iff } \forall\, a \in \alpha(M) \;.\; a \sqsubseteq imp(a) \;\; i.e., I \wedge J \wedge P \Rightarrow [op(imp(a))]\neg[a]\neg J$$

This states that any *imp* trace refinement is respected in the B machine: any sequence of operations corresponding to a matches the operation a. It is complementary to the trace notion of refinement $\sqsubseteq^T_{imp}$, which requires that only those concrete sequences of operations that correspond to abstract ones should be possible.

3.5 Traces/Divergences

Now we wish to generalise the notion of refinement so that it works for refinement in the traces/divergences model.

Definition 10 (Traces/divergences refinement with respect to imp). *If imp is an implementation mapping, then*

$$\begin{aligned} P \sqsubseteq^{TD}_{imp} P' \text{ iff } P \sqsubseteq^{T}_{imp} P' & \qquad (1)\\ \wedge\; divergences(P') \subseteq \uparrow_{\alpha P'} \Big(\bigcup_{tr \in divergences(P)} \psi(tr)\Big) & \qquad (2)\end{aligned}$$

This states that any divergence of P' must correspond to a divergence of P: given a divergent trace tr of P, $\psi(tr)$ gives the corresponding divergences of P'. Thus if event a introduces divergence, then divergence can be introduced anywhere along $imp(a)$ from the first event onwards. These are exactly the sequences in $\psi(tr)$.

Example 2. Consider $imp(a) = \langle c, d\rangle$ and $imp(b) = \langle e, f\rangle$. Consider P and P' as follows, where P diverges after $\langle a, b\rangle$, and P' diverges after $\langle c, d, e\rangle$:

$$\begin{aligned} traces(P) &= \{\langle\rangle, \langle a\rangle, \langle a, b\rangle\} \cup \{\langle a, b\rangle \frown s \mid s \in \{a, b\}^*\}\\ divergences(P) &= \{\langle a, b\rangle \frown s \mid s \in \{a, b\}^*\}\end{aligned}$$

$$\begin{aligned} traces(P') &= \{\langle\rangle, \langle c\rangle, \langle c, d\rangle, \langle c, d, e\rangle\} \cup \{\langle c, d, e\rangle \frown s \mid s \in \{c, d, e, f\}^*\}\\ divergences(P') &= \{\langle c, d, e\rangle \frown s \mid s \in \{c, d, e, f\}^*\}\end{aligned}$$

Observe that $\psi(\langle a, b\rangle) = \{\langle c, d, e\rangle, \langle c, d, e, f\rangle\}$ and so the condition in Line 2 is satisfied, and $P \sqsubseteq^{TD}_{imp} P'$.

We obtain the same result, again proved by structural induction over P, for trace divergence refinement as we did in Theorem 2 for trace refinement.

Theorem 3. $\forall\, imp, P \,.\, P \sqsubseteq_{imp}^{TD} \Theta_{imp}(P)$

The previous definitions have laid the groundwork for the following result, which is the key compositionality property we have been working towards:

Theorem 4 (Trace divergence refinement in controlled components). *If* $P \sqsubseteq_{imp}^{TD} P'$ *and* $M \sqsubseteq_{imp}^{B} M'$ *then* $P \parallel M \sqsubseteq_{imp}^{TD} P' \parallel M'$.

Proof. We know

$$traces(P') \subseteq \downarrow (\bigcup_{tr \in traces(P)} \phi(tr)) \tag{3}$$

$$divergences(P') = \emptyset = divergences(P) \tag{4}$$

$$traces(M') = (\alpha M')^* \quad where \quad \alpha M' \subseteq \alpha P' \tag{5}$$

$$traces(M) = (\alpha M)^* \quad where \quad \alpha M \subseteq \alpha P \tag{6}$$

Then consider $tr \in divergences(P' \parallel M')$. Then let tr_0 be the minimal divergent prefix of tr.

Then $tr_0 \in traces(P')$ and $tr_0 \upharpoonright \alpha M' \in divergences(M')$

$\exists\, tr'' \in traces(P).tr_0 \leq \phi(tr'')$ from (3)

Also $tr_0 \upharpoonright \alpha M' \in divergences(M')$ so $\phi(tr'') \upharpoonright \alpha M' \in divergences(M')$. Therefore $tr'' \upharpoonright \alpha M \in divergences(M)$ from Lemma 2 below.

Therefore $tr'' \in divergences(M) \cap traces(P)$ so $tr'' \in divergences(P \parallel M)$.

Lemmas 1 and 2 below are used in the proof of Theorem 4 above.

Lemma 1. *If* $M \sqsubseteq_{imp}^{B} M'$ *then* $I \wedge J \wedge [op(tr)]true \Rightarrow [op(\phi_{imp}(tr))]true$

Lemma 2. *If* $M \sqsubseteq_{imp}^{B} M'$ *and* $(\phi_{imp}(tr)) \upharpoonright \alpha M'$ *is a divergence of* M' *then* $tr \upharpoonright \alpha M$ *is a divergence of* M.

Theorem 4, unlike Theorem 1, requires the refinement relationship between machines. When only traces are considered, internal states of the machines do not affect the semantics of the parallel combination, so refinement relies purely on the CSP controllers. However, when divergences are also considered, then divergent behaviour (corresponding to an operation being called outside its precondition) is reflected in the semantics. Hence refinement of a controlled component requires that the states of the machines match up, so the concrete machine can diverge only where the abstract machine description allows it.

Example 3. Consider $M \sqsubseteq_{imp}^{B} M'$ where

- $imp(a) = \langle b, c \rangle$; $imp(w) = \langle v \rangle$; where a, b, and c are machine operations and w and v are not;
- Machine M has operation $a \mathrel{\widehat{=}} BEGIN\ nn := nn + 4\ END$;

– Machine M' has $b \mathrel{\widehat{=}} PRE\ even(mm)\ THEN\ mm := mm + 1\ END$ and $c \mathrel{\widehat{=}} PRE\ \neg even(mm)\ THEN\ mm := mm + 3\ END$.

The example shows that an event can be refined to a sequence of events. M' does contain divergences (e.g. $\langle b, b\rangle$ or $\langle b, c, c\rangle$), but the refinement of M and M' is in the context of imp so only sequences which are the image of some abstract sequence need to be considered. Therefore, we need only show that refining a by the sequence of operations $(b;c)$ is an appropriate B refinement, achieved in practice by discharging the proof obligation identified in Definition 9. An appropriate J would be $nn = mm$. We could equally have reused nn in M'. Divergent sequences of operations such as $(b;b)$ and $(b;c;c)$ are ruled out since they cannot arise from an application of imp to an abstract trace.

Consider an abstract trace $tr = \langle a, w, a\rangle$. Then $\phi(tr) = \langle b, c, v, b, c\rangle$. If $tr \upharpoonright \alpha M = \langle a, a\rangle$ is not a divergence of M, then $\phi(tr) \upharpoonright \alpha M' = \langle b, c, b, c\rangle$ is not a divergence of M' by the contrapositive of Lemma 2.

Define $P = a \rightarrow w \rightarrow P$ and $\Theta_{imp}(P) = P' = b \rightarrow c \rightarrow v \rightarrow P'$. We have $P \sqsubseteq^{TD}_{imp} P'$ from Theorem 3. Theorem 4 then yields that $P \parallel M \sqsubseteq^{TD}_{imp} P' \sqsubseteq M'$.

4 The Refinement Framework with i/o

We begin by focusing on the B framework. Our form of interface refinement in the context of operation input and output means that the input and output values across the operations need to be related.

4.1 Refining B Operations

For a given event a with $imp(a) = cs = \langle c_1, \ldots, c_n\rangle$, let in_a be the sequence of input variables to a, and out_a be the sequence of output variables for a, i.e. the declaration of a is $out_a \longleftarrow a(in_a)$. Let in_{cs} be the sequence of input variables to the collection of the c operations for $c \in cs$, and out_{cs} be the sequence of output variables for the c operations. In other words, if the c_i operations' declarations are $out_{c_i} \longleftarrow c_i(in_{c_i})$, then $out_{cs} = out_{c_1} {}^\frown \ldots {}^\frown out_{c_n}$, and $in_{c_i} = in_{c_1} {}^\frown \ldots {}^\frown in_{c_n}$. We assume that all operations have disjoint input and output variable names.

An interface refinement for a will relate the abstract and concrete input variables, and similarly with the output variables. The relationships can be formalised with a relation $r_{in,a}$ relating the abstract and concrete input variables, and a relationship $r_{out,a}$ relating the abstract and concrete output variables. These relations may be thought of as linking invariants for the inputs and for the outputs. We will use r to abbreviate the collection of all the $r_{in,a}$ and $r_{out,a}$.

We generalise Definition 9. The refinement relation is with respect both to the mapping imp and the collection of relations r:

Definition 11 (Refinement of operations within B machines). *If M and M' have linking invariant J then*

$$\begin{array}{l} M \sqsubseteq^{B}_{imp,r} M' \text{ iff} \\ \quad \forall\, a \in \alpha(M)\ ,\ r_{in,a} \wedge I \wedge J \wedge P_a \Rightarrow [op(imp(a))]\neg[a]\neg(J \wedge r_{out,a}) \end{array}$$

4.2 Examples Illustrating Aspects of Definition 11

Example 4 (Implementation modulo 5*).* The example in Figure 1 considers a change in data representation, resulting in a loss of information but in a way that allows refinement. Our single operation multiplies an input by 3 and returns the result. If we wish to refine this so that all values are modulo 5, then the refined operation may be used. This only inputs and outputs values modulo 5. The relations on inputs and on outputs capture this relationship: input of an abstract value is implemented by the input of that value modulo 5, and the resulting output will be the abstract output, modulo 5. The resulting proof obligation can be discharged to establish the refinement relationship.

```
MACHINE     Times3                          MACHINE     Times3R
OPERATIONS                                  OPERATIONS
  yy <-- triple(xx) =                         zz <-- tripleR(ww) =
      PRE xx : NAT                                PRE ww : 0..4
      THEN yy := 3 * xx                           THEN zz := (ww * 3) mod 5
      END                                         END
END                                         END
```

Fig. 1. Tripling, modulo 5

*Times*3*R* is a refinement of *Times* with $imp(triple) = \langle tripleR \rangle$ and the following definitions, which together satisfy the proof obligation of Definition 11:

$$J = true \qquad r_{in,triple} : ww = xx \, mod \, 5 \qquad r_{out,triple} : zz = yy \, mod \, 5$$

Example 5 (Change of offset). In the example of Figure 2, we change the offset of the readings, so that concrete inputs are the abstract inputs offset by +1.

```
MACHINE         Increase            MACHINE         IncreaseR
VARIABLES       total               VARIABLES       totalR
INVARIANT       total : NAT         INVARIANT       totalR : NAT
INITIALISATION  total := 0                             & totalR = total
OPERATIONS                          INITIALISATION  totalR := 0
  add(xx) =                         OPERATIONS
      PRE xx : NAT                    addR(ww) =
      THEN total := total + xx            PRE ww : NAT
      END                                 THEN totalR := totalR + (ww - 1)
END                                       END
                                    END
```

Fig. 2. Change of offset I

An abstract input value xx is implemented by the concrete value $xx + 1$. This is captured in the relation $r_{in,add}$.

IncreaseR is a refinement of *Increase*, under the following definitions:

$$imp(add) = \langle addR \rangle \qquad J : totalR = total \qquad r_{in,add} : xx = ww - 1$$

The proof obligation of Definition 11 is met by these definitions. The steps are as follows:

$$\begin{aligned}
&[addR]\neg[total := total + xx]\neg(J \wedge r_{out,add}) \\
&= [addR](total + xx = totalRR) \\
&= ww \in NAT \wedge (total + xx = totalR + ww - 1) \\
&\Leftarrow xx \in NAT \wedge xx = ww - 1 \wedge total = totalR \qquad (7) \\
&= P_{add} \wedge r_{in,add} \wedge J
\end{aligned}$$

Example 6 (Change of offset). The example in Figure 3 is similar to the previous example, except that the concrete inputs are the abstract inputs offset by -1.

```
MACHINE         Increase          MACHINE         IncreaseR
VARIABLES       total             VARIABLES       totalR
INVARIANT       total : NAT       INVARIANT       totalR : NAT
INITIALISATION  total := 0                          & totalR = total
OPERATIONS                        INITIALISATION  totalR := 0
  add(xx) =                       OPERATIONS
      PRE xx : NAT                  addR(ww) =
      THEN total := total + xx          PRE ww : NAT
      END                               THEN totalR := totalR + (ww + 1)
END                                     END
                                  END
```

Fig. 3. Change of offset II

The change of offset is captured in the relation $r_{in,add}$. One might hope that the following definitions would show that *IncreaseR* is a refinement of *Increase*:

$$imp(add) = \langle addR \rangle \qquad J : totalR = total \qquad r_{in,add} : xx = ww + 1$$

However, the proof obligation of Definition 11 is not met by these definitions, and in particular the implication in Line 7 does not carry through, since an abstract input $xx = 0$ cannot be matched by any natural number ww. Note that if the precondition on the concrete operation allowed ww also to range over negative integers, then the proof obligation would be met: clearly the abstract value 0 is represented by -1.

Examples 5 and 6 together illustrate the delicate relationship between what is required by the refinement and what is allowed by the abstract machine. We see that whenever the abstract operation is enabled with a particular input, then the refinement must also be enabled with a related input value. However, we see

```
MACHINE         Sensor
VARIABLES       tt, pp
INVARIANT       tt : NAT
                 & pp : NAT
INITIALISATION  tt :: NAT
                 || pp :: NAT
OPERATIONS
  update(dt,dp) =
      PRE dt : NAT & dp : NAT
      THEN tt := tt + dt
            || pp := pp + dp
      END
END
```

```
MACHINE        SensorR
VARIABLES      rrR, ppR
INVARIANT      ttR : NAT & ppR : NAT
                & ttR = tt & ppR = pp
INITIALISATION  ttR :: NAT
                 || ppR :: NAT
OPERATIONS
  updatet(dt1) =
      PRE dt1 : NAT
      THEN ttR := ttR + dt1
      END;
  updatep(dp1) =
      PRE dp1 : NAT
      THEN ppR := ppR + dp1
      END
END
```

Fig. 4. Distributing inputs

from Example 5 that the converse is not the case: the concrete input 0 there does not correspond to any abstract input. The abstract machine imposes no requirements on the refinement behaviour for that input value: it corresponds to a value that is outside the abstract precondition.

Example 7 (distributing inputs)
In Figure 4, *SensorR* is a refinement of *Sensor* with the following definitions:

$$imp(update) = \langle updatet, updatep \rangle \qquad r_{in,update} : dt = dt1 \wedge dp = dp1$$

Observe that the proof obligation requires only that the abstract and refined machine states match at the end of the sequence of concrete operations. The refinement machine will pass through states that need not match the abstract state.

5 Trace Refinement with i/o

Given an implementation mapping *imp* and relations $r_{in,a}$, $r_{out,a}$, we can define a refinement relation on processes that incorporates the input and output values.

Given a particular $r_{in,a}$ (as used in the machine refinement), and where $imp(a) = \langle c_1, \ldots, c_n \rangle$ we will define the sequences of concrete events with their inputs and outputs, associated with an abstract i/o event $a.v.w$, where v is the inputs to a, and w is the outputs. The mapping *imp* lifts to a mapping *imp'* which gives the set of all sequences corresponding to a particular i/o event:

Definition 12

$$imp'(a.v.w) = \{\langle c_1.v_1.w_1, \ldots, c_n.v_n.w_n \rangle \mid r_{in,a}(v, v_1, \ldots, v_n) \wedge r_{out,a}(w, w_1, \ldots, w_n)\}$$

The function ϕ then generalises as follows:

$$\phi_{imp,r}(\langle\rangle) = \{\langle\rangle\} \qquad \phi_{imp,r}(\langle a.v.w\rangle \frown tr) = imp'(a.v.w) \frown \phi_{imp,r}(tr)$$

This supports the natural definition of trace refinement: that every trace of P' should arise from some trace of P.

Definition 13 (trace refinement relative to *imp* and *r*)
$P \sqsubseteq^T_{imp,r} P'$ *iff* $traces(P') \subseteq \downarrow(\bigcup_{tr \in traces(P)} \phi_{imp,r}(tr))$

We have already identified a notion of refinement for processes, and one for machines in terms of relationships between their operations. We are aiming for the following compositionality result, which is an extension of Theorem 4 in the context of the relation r on i/o:

Conjecture 1. If $P \sqsubseteq^T_{imp,r} P'$ and $M \sqsubseteq^B_{imp,r} M'$ then $P \parallel M \sqsubseteq^T_{imp,r} P' \parallel M'$.

Note that the traces of machines M are no longer all possible traces (as they are without i/o), since they constrain the possible outputs. Hence the conjecture takes the machine traces into account, since they restrict the overall behaviour.

This notion of refinement is not reflexive: $P \sqsubseteq_{imp,r} P$ does not hold in general, because the inputs and outputs may change (even where *imp* is the identity function). Hence a combination $P \parallel M$ will not be refined simply by refining M; the controller will need to be refined as well.

6 Discussion and Related Work

In this paper we presented the theoretical framework to support the refinement of an abstract event with a sequence of concrete events within the CSP$\|B$ framework. From this point of view the important result is Theorem 4. We also described what it means to distribute inputs and outputs across the concrete sequence of operations, and showed how the type of the inputs and outputs can also be refined. Natural extensions to the work are consideration of failures information, and refinement of events by processes, allowing nondeterminism. We were not able to consider them in this paper for reasons of space.

In [7], Derrick and Boiten present a theory for non-atomic refinement using Z. They also support the refinement of an abstract operation with a sequence of concrete operations. Our motivation is the same as theirs: the precise structure of an implementation may not be known at the abstract level and we need to provide a way of being able to introduce more detail at the concrete level. We can also split a collection of inputs and/or outputs across a number of operations. The difference with our work is that the sequences of operations we need to consider are defined within a CSP controller and the implementation mapping between abstract and concrete operations is explicitly described.

Derrick and Boiten also consider a notion of I/O refinement in [5, Chapter 10]. They establish conditions for changing the I/O within single operations to provide a refinement, using input and output transformers, which play a similar role to our relations $r_{in,a}$ and $r_{out,a}$. In [8] Derrick and Wehrheim bring together the ideas from [7] and [5] and refine atomic operations by sequences of operations together with I/O refinement. Their approach is entirely state-based, which makes the handling of sequences of operations more difficult, and the authors state in their conclusions that the combination with process algebra remains to be investigated. This paper does combine the state-based view with a process algebra, giving explicit and natural descriptions of control in specifications, and so handling the refining sequences of operations more easily.

In Event-B [2], a refinement of an event, e.g., a can be achieved using several events (at least one), one of which must be the refinement of the original a event. Any new events must be a refinement of *Skip*. Event-B refinement proof obligations ensure that new events do not cause infinite internal behaviour. Furthermore, new events can occur non-deterministically, provided their guards are true, i.e., Event-B does not require an explicit scheduler. We have shown how to refine an event (which may have a corresponding B operation) with a single sequence of events (again with underlying B operations) and thus an explicit schedule must be provided in the refinement. This may be restrictive when there are several scheduling possibilities. However, if the scheduler is known in advance then we provide an explicit way of describing it in a refinement. Also, we do not require that one event is a refinement of the original event. What we require is that a sequence of events is an appropriate refinement of an abstract event. Our refinement also allows I/O refinement and type refinement of the inputs and the outputs; recent research in Event-B is also examining how to include I/O parameters in events [6].

Our approach to traces and trace divergences event refinement bears some resemblance to the approaches to action refinement in process algebras developed in the 1980's and early 1990's, see e.g. [3], where single events are refined by more complex behaviour. However, the focus then was within pure process algebra, and with more intricate semantics. In contrast, our emphasis is on developing an approach which integrates with state-based components, in our case B-machines, and it is this emphasis that has driven the development of the approach presented in this paper.

In our recent work we have continued to generalise the results in order to support refining events to sets of sequences of events and to processes.

Acknowledgments. Thanks to Thai Son Hoang for discussions related to Event-B. Thanks also to the IFM reviewers for their insightful and helpful comments.

References

1. Abrial, J.-R.: The B Book: Assigning programs to meanings. Cambridge University Press, New York (1996)
2. Abrial, J.-R.: Modelling in Event-B: System and Software Engineering. Cambridge University Press (in preparation)
3. Aceto, L.: Action Refinement in Process Algebras. Cambridge University Press, Cambridge (1992)

4. Bergstra, J.A., Ponse, A., Smolka, S.A. (eds.): Handbook of Process Algebra. North-Holland, Amsterdam (2001)
5. Boiten, E., Derrick, J.: Refinement in Z and Object-Z: Foundations and Advanced Applications. Springer, Heidelberg (2001)
6. Butler, M.: Personal communication (September 2008)
7. Derrick, J., Boiten, E.: Non-atomic refinement in Z. In: Woodcock, J.C.P., Davies, J., Wing, J.M. (eds.) FM 1999. LNCS, vol. 1709, pp. 1477–1496. Springer, Heidelberg (1999)
8. Derrick, J., Wehrheim, H.: Using coupled simulations in non-atomic refinement. In: ZBB (2003)
9. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
10. Morgan, C.: Of wp and CSP. In: Beauty is our business: a birthday salute to E. W. Dijkstra, pp. 319–326 (1990)
11. Schneider, S.: Concurrent and Real-Time Systems: the CSP Approach. Wiley, Chichester (1999)
12. Schneider, S.: The B-Method: an introduction. Palgrave (2001)
13. Schneider, S., Pizarro, D., Treharne, H.: The futuretech demonstrator, Future Technologies for System Design Technical Report, University of Surrey (2008)
14. Schneider, S., Treharne, H.: CSP theorems for communicating B machines. Formal Asp. Comput. 17(4), 390–422 (2005)

CSP with Hierarchical State

Robert Colvin and Ian J. Hayes

The University of Queensland,
ARC Centre for Complex Systems,
School of Information Technology and Electrical Engineering,
Brisbane, Australia

Abstract. The process algebra CSP is designed for specifying interactions between concurrent systems. In CSP, and related languages, concurrent processes synchronise on common events, while the internal operations of the individual processes are treated abstractly. In some contexts, however, such as when modelling systems of systems, it is desirable to model both interprocess communications as well as the internal operations of individual processes. At the implementation level, shared state is often the method of communication between processes, and tests and updates of local state are used to implement internal operations. In this paper we propose an extension of the CSP language which maintains CSP's core elegance in specifying process synchronisation, while also allowing state-based behaviour. State is treated *hierarchically*, allowing (nested) declarations of local and shared variables. The state can be accessed and modified using a refinement calculus-style *specification command*, which may be optionally paired with event synchronisation. The semantics of the extended language, preserves the original CSP rules. The approach we present is novel in that state is part of the process, rather than a meta-level construct appearing only in the rules.

1 Introduction

The process algebra CSP [7] is a language designed for specifying concurrent systems in which processes interact by synchronising and exchanging information through a set of common events. Because the focus is on interactions, the internal operations of a process are treated abstractly. In this paper we give an extension of CSP to include a construct for declaring state, and a general construct for testing and updating the state. The state can be declared local to a single sequential process, or shared between concurrent processes. The integration of state-based constructs with CSP enables the internal operations of a process to be specified in a familiar, imperative programming style, and modelling of shared-state systems. The extension is designed so that CSP's core elegance in specifying interprocess communication is maintained, and therefore obeys the following constraints: it is "lightweight", in that it includes only a few straightforward language extensions which do not affect existing constructs (syntactic preservation), and therefore remains faithful to the original style of CSP; all existing CSP operational laws remain valid (semantic preservation); it allows both local state as well as shared state in a hierarchical way; and it allows combining state operations with a single synchronisation.

In Sect. 2 we review CSP and present the operational semantics of a subset of the language. In Sect. 3 we extend CSP with a construct for declaring state, and commands for testing and updating state. In Sect. 4 we generalise tests and updates to any relationship between pre- and post-states, and allow them to be combined with event synchronisation in a single atomic action. Related work is discussed in Sect. 5 and we conclude in Sect. 6.

2 Review of CSP

2.1 Syntax

CSP is a process algebra which allows concurrent processes to communicate synchronously via shared events. We base our understanding of CSP on the book by Hoare [7], and present its meaning via an operational semantics in the style of Roscoe [11] and Schneider [12].

Processes interact via a set of events, *Event*. In addition, there are two special events, τ, representing an *internal* (unobservable) event, and $\checkmark$, representing successful termination. A subset of the syntax of a process in CSP is summarised below.

$$\begin{aligned} P ::= {} & (a \rightarrow P) \mid (P_1 \sqcap P_2) \mid (P_1 \mathbin{[\!]} P_2) \mid (P_1 \parallel P_2) \mid (P \backslash A) \mid (P_1 \,;\, P_2) \mid \\ & \mathit{SKIP}_A \mid \mathit{STOP}_A \end{aligned} \tag{1}$$

An *event prefix* process $a \rightarrow P$, where $a \in \mathit{Event}$, is one that synchronises on event a before behaving as process P. An *internal choice* between P and Q, written $P \sqcap Q$, nondeterministically chooses between P and Q, without reference to a particular event. An *external choice* between processes P and Q is given by $P \mathbin{[\!]} Q$. Whichever process is the first to perform a synchronisation event with the environment becomes active. Concurrency is written by $P \parallel Q$, which states that the two processes operate in parallel, synchronising on shared events and interleaving non-shared events. A set of events $A \subseteq \mathit{Event}$ within P may be "hidden", written $P \backslash A$, so that any events in A are not visible externally to P (these become *internal* steps of $P \backslash A$). A sequential composition $P ; Q$ behaves as P until P terminates, after which it behaves as Q. The process SKIP_A has only one possible behaviour, which is to terminate successfully and take no further action. The process STOP_A has no behaviour – it may never synchronise or take any other action. Both SKIP_A and STOP_A are parameterised by a set of events A, which forms their *alphabet*, described below. In general, processes may also be parameterised by values, examples of which are given below.

Events can contain values, e.g., $c.v$, where c is a *channel* name and v is a value. Channels are used for passing information from one process to another. By convention, the sending process $c.v \rightarrow P$ is written $c!v \rightarrow P$, and the receiving process is written $c?x : T \rightarrow P(x)$, which represents a process that engages in any event $c.v$ for $v \in T$, then behaves as $P(x)$ (a process dependent on x). In this paper, to avoid distractions associated with type systems, we assume all values are of a universal type *Val*, and hence will omit the type qualifier T.

CSP includes a range of other operators, but for the purposes of this paper we take the above subset as core. As an example, consider the specification of a queue in CSP.

$$\begin{aligned} &Qu(\langle\rangle) \mathrel{\widehat=} enq?x \rightarrow Qu(\langle x\rangle) \\ &Qu(\langle y\rangle \frown q) \mathrel{\widehat=} (enq?x \rightarrow Qu(\langle y\rangle \frown q \frown \langle x\rangle)) \mathbin{[\!]} (deq!y \rightarrow Qu(q)) \end{aligned} \tag{2}$$

The state of the queue is maintained as a parameter to the process. Values are enqueued via channel *enq* and dequeued on *deq*. We have followed Schneider's [12] style of separating the empty and non-empty cases of the queue parameter. This is required because the *deq* channel is not available if the queue is empty.

The set of events a process P may engage in is given by its alphabet, $\alpha(P) \subseteq Event \cup \{\checkmark\}$. The alphabet of a process must satisfy the equations below.

$$\begin{array}{rlrl} \alpha(a \rightarrow P) &= \alpha(P) \text{ where } a \in \alpha(P) & \alpha(P \backslash A) &= \alpha(P) \setminus A \\ \alpha(P \sqcap Q) &= \alpha(P) = \alpha(Q) & \alpha(P \,;\, Q) &= \alpha(P) \cup \alpha(Q) \\ \alpha(P \mathbin{[\!]} Q) &= \alpha(P) = \alpha(Q) & \alpha(SKIP_A) &= A \cup \{\checkmark\} \\ \alpha(P \parallel Q) &= \alpha(P) \cup \alpha(Q) & \alpha(STOP_A) &= A \end{array} \tag{3}$$

Unless otherwise specified, we assume the alphabet of a process is the minimum required to satisfy the above rules. For instance, a non-empty queue can engage in all *enq* and *deq* events, while an empty queue can immediately engage in *enq* events, followed by all events in the alphabet of a non-empty queue. Define $c.Val \mathrel{\widehat=} \{v : Val \bullet c.v\}$, i.e., $c.Val$ is the set of events formed from channel c and a value. Then for any sequence of values, q, the definition $\alpha(Qu(q)) = enq.Val \cup deq.Val$ is valid because it satisfies the equations given in (3).

As a more complex process example, consider a system which is formed from two processes, S and R, which communicate via a buffer implemented as a queue (process Qu). S generates values for the queue and R reads those values and performs some actions on them. Assume the existence of a process *Produce* which generates a value x. Process S repeatedly starts process $Produce(x)$ and puts x on the queue. Process R reads value y from the queue and then performs some actions via process $Consume(y)$. The whole system, *Sys*, is formed by running S and R in parallel with $Qu(\langle\rangle)$, and hiding the communication events.

$$\begin{aligned} S &\mathrel{\widehat=} (Produce(x) \,;\, (enq!x \rightarrow S)) \\ R &\mathrel{\widehat=} deq?y \rightarrow (Consume(y) \,;\, R) \\ Sys &\mathrel{\widehat=} (S \parallel R \parallel Qu(\langle\rangle)) \backslash (enq.Val \cup deq.Val) \end{aligned} \tag{4}$$

Process *Sys* hides all communication on channels *enq* and *deq*. Processes external to *Sys* will not see any of the events associated with *enq* or *deq* – all internal communication will appear as a step in τ. (*Sys* is not a recursive process and hence may declare the hiding internally.)

$$(a \rightarrow P) \xrightarrow{a} P \tag{5}$$

$$(P \sqcap Q) \xrightarrow{\tau} P \tag{6}$$
and similarly for Q.

$$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q'}{P \parallel Q \xrightarrow{a} P' \parallel Q'} \tag{7}$$

$$\frac{P \xrightarrow{\mu} P' \quad \mu \notin \alpha(Q)}{P \parallel Q \xrightarrow{\mu} P' \parallel Q} \tag{8}$$
and similarly for Q.

$$SKIP_A \xrightarrow{\checkmark} STOP_A \tag{9}$$

$$\frac{P \xrightarrow{\tau} P'}{P \mathbin{[\!]} Q \xrightarrow{\tau} P' \mathbin{[\!]} Q} \qquad \frac{P \xrightarrow{a} P'}{P \mathbin{[\!]} Q \xrightarrow{a} P'} \tag{10}$$
and similarly for Q.

$$\frac{P \xrightarrow{a} P' \quad a \in A}{P \backslash A \xrightarrow{\tau} P' \backslash A} \qquad \frac{P \xrightarrow{\mu} P' \quad \mu \notin A}{P \backslash A \xrightarrow{\mu} P' \backslash A} \tag{11}$$

$$\frac{P \xrightarrow{\mu} P' \quad \mu \neq \checkmark}{P \mathbin{;} Q \xrightarrow{\mu} P' \mathbin{;} Q} \qquad \frac{P \xrightarrow{\checkmark} P'}{P \mathbin{;} Q \xrightarrow{\tau} Q} \tag{12}$$

Fig. 1. Rules for CSP

2.2 Operational Semantics

The operational semantics for CSP operators is given in Fig. 1, and follows the style of Schneider [12]. Internal transitions are labelled with the internal event τ, and unless otherwise stated, transitions labelled with a are valid for $a \in \mathit{Event} \cup \{\checkmark\}$. Transitions labelled with μ apply for $\mu \in \mathit{Event} \cup \{\checkmark, \tau\}$.

Rule (5) states that a prefixed process $a \rightarrow P$ may take a step in a then behave as P. (Recall that a prefix a is a member of *Event*, i.e., cannot be τ or $\checkmark$.) Rule (6) states that an internal choice between P and Q may evolve to either process, without any visible action. Rule (7) states that P and Q may synchronise on event a if both are able to do so. Rule (8) states that P may evolve by interleaving an event μ that is not in the alphabet of Q. Rule (9) states that the special process $SKIP_A$ transitions in the $\checkmark$ event and then takes no further action. There are no rules for $STOP_A$ – it cannot engage in any actions. We follow the convention of omitting the alphabet subscript on $SKIP_A$ and $STOP_A$ when it is clear from context. Rule (10) states that P in an external choice may take an internal step without affecting the choice itself. Alternatively, if P may take a step with event a then the choice may be made in P's favour. Symmetric rules holds for Q. Rule (11) states that if P may take a step in a and a is hidden, then $P \backslash A$ may take an internal step. Alternatively, $P \backslash A$ may take a step in a if a is not hidden. Rule (12) states that until P terminates, $P \mathbin{;} Q$ behaves as P, after which it behaves as Q.

As an example, an initially empty Qu process can take the following steps.

$$Qu(\langle\rangle) \xrightarrow{enq.v} Qu(\langle v\rangle) \xrightarrow{enq.w} Qu(\langle v, w\rangle) \xrightarrow{deq.v} Qu(\langle w\rangle) \xrightarrow{deq.w} Qu(\langle\rangle)$$

These steps are justified by Rules (5) and (10). If channels *enq* and *deq* were hidden, each transition label above would be τ (Rule (11)). Without the hiding, the queue process could synchronise and exchange values with a concurrent process which is listening to the *enq* and *deq* events via Rule (7), such as processes S and R in (4).

3 CSP Extended with State

We extend the CSP language with operators for testing and updating state, and call the new language CSP$_\sigma$. We assume the existence of a set of variable identifiers, *Var*, a set of values of variables, *Val*, and define a *state* as a partial mapping from variables to values, $\mathit{state} \mathrel{\widehat=} \mathit{Var} \mathrel{\ooalign{$\rightarrow$\cr\hfil$\mapstochar$\hfil}} \mathit{Val}$. We use σ to denote such partial states; on occasion we refer to a state which is total on *Var*, and we denote such states by Σ.

3.1 Syntax

Three new operators are added to the language, as shown below, where σ is a state, g is a predicate, x is a variable, and E is an expression[1].

$$P ::= (\mathbf{st}\ \sigma \bullet P) \mid ([g] \rightarrow P) \mid ((x \coloneqq E) \rightarrow P) \mid \ldots \tag{13}$$

where '...' includes the operators for CSP from (1). Guards and updates do not appear in process event alphabets.

A local state process $(\mathbf{st}\ \sigma \bullet P)$ defines the variables in the domain of σ to be local to P, with their value in σ giving their initial value. Updates to and accesses of variables in the domain of σ are hidden from external observers, and hence, the local state hides variables in the same way that events in A are hidden by $P \backslash A$. Local states may be declared hierarchically (nested), e.g., $(\mathbf{st}\ \sigma_1 \bullet (\mathbf{st}\ \sigma_2 \bullet P))$, where expressions in P may contain free variables which are in the domain of σ_1 or σ_2. If a variable x appears in both σ_1 and σ_2, the value for x in σ_2 overrides the value in σ_1. We call the combination of all local states for some process its *context*.

A *guard prefix* process $[g] \rightarrow P$ is blocked from proceeding until predicate g is evaluated to true in the current context, after which it behaves as P. An *update prefix* process $(x \coloneqq E) \rightarrow P$ updates the variable x to the expression E evaluated in the current context, then behaves as P. We use the term *action* to cover everything that may appear on the left-hand side of a prefix, i.e., events, guards and updates.

As an example, consider the (at this stage flawed) specification of a queue in CSP$_\sigma$. We define a process Q which handles messages along channels *enq* and *deq* as before, explicitly updating queue variable q. The variable q is declared locally to process Qu.

$$\begin{aligned}
Qu &\mathrel{\widehat=} (\mathbf{st}\ \{q \mapsto \langle\rangle\} \bullet Q) \\
Q &\mathrel{\widehat=} \begin{array}{l} \quad \mathit{enq}?x \rightarrow (q \coloneqq q \frown \langle x \rangle) \rightarrow Q \\ \,[\!]\ [q \neq \langle\rangle] \rightarrow \mathit{deq}!\mathit{head}(q) \rightarrow (q \coloneqq \mathit{tail}(q)) \rightarrow Q \end{array}
\end{aligned} \tag{14}$$

After an *enq*?x event, q is explicitly updated and the process repeats. If q is nonempty Q may participate in *deq* events. However, once the second branch is chosen, Q will refuse *enq* events, which may lead to deadlock if the environment

[1] To avoid distractions we assume all expressions are well defined.

is not offering *deq*. The desired behaviour is that the event *deq* may occur, and the second branch selected, only if $q \neq \langle\rangle$. To achieve this we must be able to atomically combine guards with events; this will be explored in Sect. 4. Processes S, R and *Sys* from (4) may be defined similarly in CSP$_\sigma$, except the *Qu* process does not need a parameter.

Computation sequences can be specified in CSP$_\sigma$, as shown by the following process $Sum(X)$ which calculates the sum of the natural numbers up to X using local variables s and i, and then writes the result to non-local variable x.

$$\begin{aligned} Sum(X) &\mathrel{\widehat=} (\mathbf{st}\ \{i \mapsto 1, s \mapsto 0\} \bullet S(X)) \\ S(X) &\mathrel{\widehat=} \begin{array}{l} [i \leq X] \rightarrow (s \coloneqq s + i) \rightarrow (i \coloneqq i + 1) \rightarrow S(X) \\ \mathbin{[\![} [i > X] \rightarrow (x \coloneqq s) \rightarrow \mathit{SKIP} \end{array} \end{aligned} \tag{15}$$

To an external observer, all of the steps prior to the final copy to x are internal since they test and update only the local variables i and s.

3.2 Operational Semantics

Our novel approach to defining the operational semantics of guards and updates is to introduce them as transition labels which are "hidden" by the closest declaring state, in the same way that events may be hidden in CSP. However, because states may only hide some of the free variables referenced by a guard or update, such states only partially hide those transitions. For example, consider the following transitions of process P prefixed by a guard, within a state which maps i to 1. The local state $\{i \mapsto 1\}$ 'hides' i from external observers.

$$(\mathbf{st}\ \{i \mapsto 1\} \bullet [i \leq 5] \rightarrow P) \xrightarrow{\top} (\mathbf{st}\ \{i \mapsto 1\} \bullet P) \tag{16}$$

$$(\mathbf{st}\ \{i \mapsto 1\} \bullet [i \leq x] \rightarrow P) \xrightarrow{[1 \leq x]} (\mathbf{st}\ \{i \mapsto 1\} \bullet P) \tag{17}$$

$$(\mathbf{st}\ \{i \mapsto 1\} \bullet [y \leq x] \rightarrow P) \xrightarrow{[y \leq x]} (\mathbf{st}\ \{i \mapsto 1\} \bullet P) \tag{18}$$

In (16) the transition label is $\top$, which plays a similar role to τ. The guard trivially evaluates to true in the local state, so to an external observer some internal step is taken. In (17) the guard accesses non-local variable x. The externally observable behaviour of this process is that it will evolve to P if $x \geq 1$. The predicate has been partially instantiated according to the local state. In (18) the local state has no effect on the guard: its progress is dependent on non-local variables and hence is externally visible (via the transition label). A process $(\mathbf{st}\ \{i \mapsto 1\} \bullet [i > 5] \rightarrow P)$ cannot transition at all since the guard does not hold in the local context.

Now consider the following transitions of process P prefixed by an update of variable s.

$$(\mathbf{st}\ \{i \mapsto 1\} \bullet s \coloneqq 0 \rightarrow P) \xrightarrow{s \coloneqq 0} (\mathbf{st}\ \{i \mapsto 1\} \bullet P) \tag{19}$$

$$(\mathbf{st}\ \{s \mapsto 1\} \bullet s \coloneqq 0 \rightarrow P) \xrightarrow{\top} (\mathbf{st}\ \{s \mapsto 0\} \bullet P) \tag{20}$$

$$(\mathbf{st}\ \{i \mapsto 1\} \bullet s \coloneqq i \rightarrow P) \xrightarrow{s \coloneqq 1} (\mathbf{st}\ \{i \mapsto 1\} \bullet P) \tag{21}$$

Transition (19) describes an update to a non-local variable, in which the update expression is independent of the local state. Transition (20) describes an update of a local variable. The process evolves to P with s updated locally to 0. This is an internal transition and hence labelled with $\top$. In (21) the local context does not include s, but does include a variable in the update expression. Since i is mapped to 1 locally, to an external observer the process appears as an update of s to 1. We consider more complex update examples below.

Before giving the formal rules for state-based constructs we define some notation. For an expression E and state σ, $E[\sigma]$ represents E with its free variables that are in the domain of σ replaced by their value in σ. For instance, if E is the boolean expression $(x = y + 1)$ and σ is $\{x \mapsto 5\}$, $E[\sigma]$ is $(5 = y + 1)$. If E is a predicate, as in the above example, then $[\![E]\!]$ is true if and only if E holds for all values of all its free variables, i.e., $(\forall \Sigma \bullet E[\Sigma])$. (Recall that Σ is a state total on Var.) We write $\mathbf{sat}([g])$ if g is satisfiable for *some* values of its free variables, i.e., $(\exists \Sigma \bullet g[\Sigma])$. Note that $\mathbf{sat}([g]) = \neg[\![\neg g]\!]$.

As foreshadowed, we extend the set of possible transition labels to include SCmd, which contains guards and updates. A transition $P \xrightarrow{[g]} P'$ says that P can evolve to P' if the predicate g is true, and a transition $P \xrightarrow{x \coloneqq E} P'$ says that P evolves P' and has the effect of updating x to E. The transition $\top$ is a member of SCmd: it represents a transition in guard $[g]$ where $[\![g]\!]$. It is the state-based equivalent of τ, but note that $\top \in \mathit{SCmd}$, whereas $\tau \notin \mathit{Event}$. This makes the definition of the rules more compact, since a transition which is allowable in every state ($\top$) is just a special case. By abuse of notation we treat $\top$ as a single entity, although it in fact represents a set of transitions (e.g., $[\mathit{true}], [1 > 0], [x = x]$, etc.).

If we allow μ to also range over SCmd actions, then all of the rules in Fig. 1 still hold. The only construct from (1) which needs an additional rule to handle SCmd transitions is external choice – see Rule (23), which is the state-based equivalent of Rule (10).

Fig. 2 contains transition rules for the extended set of constructs and transition labels. Rule (22) defines the transitions for guards and updates in a similar manner to event prefixing (Rule (5)). Rule (24) states that $(\mathbf{st}\ \sigma \bullet P)$ transitions in $\tau, \checkmark$ or event a if P does. We allow event a to reference state variables (if a represents the passing of information along a channel), therefore a must be (partially) instantiated by the local state. For instance,

$$(\mathbf{st}\ \{x \mapsto 1\} \bullet c!x \to P) \xrightarrow{c.1} (\mathbf{st}\ \{x \mapsto 1\} \bullet P)$$

Rule (25) states that $(\mathbf{st}\ \sigma \bullet P)$ may transition in guard $[g[\sigma]]$ if P may take a transition in $[g]$, and $[g[\sigma]]$ is satisfiable. This proviso ensures that guards that cannot evaluate to true cannot transition. The rule may be compared to Rule (11), in the sense that variables that occur in the domain of σ are replaced in g by their local value in σ. If g only contains free variables that occur in the domain of σ, then $g[\sigma]$ may be evaluated locally: it will either evaluate to true ($[\![g[\sigma]]\!] = \mathit{true}$), in which case the transition can always occur, or it will evaluate to false, and no transition is possible (since $\mathbf{sat}([g[\sigma]])$ will not hold).

$$([g] \to P) \xrightarrow{[g]} P \qquad ((x \coloneqq E) \to P) \xrightarrow{x \coloneqq E} P \tag{22}$$

$$\frac{P \xrightarrow{\top} P'}{P \parallel Q \xrightarrow{\top} P' \parallel Q} \qquad \frac{P \xrightarrow{s} P' \quad s \in \mathit{SCmd} \setminus \{\top\}}{P \parallel Q \xrightarrow{s} P'} \tag{23}$$

and similarly for Q.

$$\frac{P \xrightarrow{\mu} P'}{(\mathbf{st}\ \sigma \bullet P) \xrightarrow{\mu[\sigma]} (\mathbf{st}\ \sigma \bullet P')} \tag{24}$$

$$\frac{P \xrightarrow{[g]} P' \quad \mathbf{sat}([g[\sigma]])}{(\mathbf{st}\ \sigma \bullet P) \xrightarrow{[g[\sigma]]} (\mathbf{st}\ \sigma \bullet P')} \tag{25}$$

$$\frac{P \xrightarrow{x \coloneqq E} P' \quad x \notin \operatorname{dom} \sigma}{(\mathbf{st}\ \sigma \bullet P) \xrightarrow{x \coloneqq E[\sigma]} (\mathbf{st}\ \sigma \bullet P')} \tag{26}$$

$$\frac{P \xrightarrow{x \coloneqq E} P' \quad x \in \operatorname{dom} \sigma \quad v \in \mathit{Val} \quad \mathbf{sat}([E[\sigma] = v])}{(\mathbf{st}\ \sigma \bullet P) \xrightarrow{[E[\sigma]=v]} (\mathbf{st}\ \sigma \oplus \{x \mapsto v\} \bullet P')} \tag{27}$$

Fig. 2. Rules for guards and updates

The example transitions (16)–(18) may all be justified by applying Rule (25) and Rule (22).

Rule (26) states $(\mathbf{st}\ \sigma \bullet P)$ may transition in "$x \coloneqq E[\sigma]$" if P may transition in "$x \coloneqq E$" and x is not local to σ. In this case, the expression E is partially instantiated with respect to σ, and the local state is not affected. Examples of this were given in (19) and (21).

Rule (27) captures the case where the updated variable x is local to P. This case is complicated by the possibility that the value of E may not be determined solely by σ, that is, when E contains variables not in the domain of σ. The rule therefore describes many possible transitions, one for each possible value of v such that $(E[\sigma] = v)$ is satisfiable. In each such transition, the local state is updated so that x is mapped to v. Importantly, the transition label $[E[\sigma] = v]$ below the line is a guard, whereas above the line the label $x \coloneqq E$ is an update. The labelling ensures that the value v chosen for x is consistent with the context. The updated state is described notationally by $\sigma \oplus \{x \mapsto v\}$; more generally, for functions f and g, f overridden by g, $f \oplus g$, is a function which returns $g(x)$ for elements in the domain of g, and $f(x)$ otherwise.

Transition (20) given earlier is a simple example of the application of (27) where we make the obvious choice of 0 for v, since $E[\sigma]$ evaluates to 0, and therefore $[E[\sigma] = v] = [0 = 0] = \top$. Given below is a set of transitions for the more complex case where the updated variable is local but the expression E is not.

$$(\mathbf{st}\ \{s \mapsto 1\} \bullet s \coloneqq s + i \to P) \xrightarrow{[i=0]} (\mathbf{st}\ \{s \mapsto 1\} \bullet P) \tag{28}$$

$$(\mathbf{st}\ \{s \mapsto 1\} \bullet s \coloneqq s + i \to P) \xrightarrow{[i=1]} (\mathbf{st}\ \{s \mapsto 2\} \bullet P) \tag{29}$$

$$(\mathbf{st}\ \{s \mapsto 1\} \bullet s \coloneqq s + i \to P) \xrightarrow{[i=2]} (\mathbf{st}\ \{s \mapsto 3\} \bullet P) \tag{30}$$

In these cases we cannot locally determine the value to which s must be updated, since the update expression accesses non-local variable i. Locally, therefore, there are many possible transitions, one for each $v \in \mathit{Val}$ to which s can be updated (we have shown only the transitions for $v = 1, v = 2, v = 3$). However, in practice, only one transition will be possible for a given context. In this case, that will be the transition in which v has the value of $1 + i$ in context.

For instance, the process can evolve to P with $s = 2$ in the local state only if $i = 1$ in the context (29). Hence, consider an outer context of P in which i has the value 1.

$$\begin{array}{l}(\mathbf{st}\ \{i \mapsto 1\} \bullet (\mathbf{st}\ \{s \mapsto 1\} \bullet s \coloneqq s + i \rightarrow P)) \xrightarrow{\top} \\ (\mathbf{st}\ \{i \mapsto 1\} \bullet (\mathbf{st}\ \{s \mapsto 2\} \bullet P))\end{array}$$

The assignment $s \coloneqq s + i$ is completely determined by the context provided by the outer state $\{i \mapsto 1\}$, and hence always transitions (in $\top$). The effect is to update the value of s within the inner state. By Rule (25), transitions (28) and (30) are not possible when the outer context determines $i = 1$, since, for example, $\mathbf{sat}([(i = 0)[\{i \mapsto 1\}]])$ does not hold.

Below is the execution of program $\mathit{Sum}(2)$ from (15).

$$\begin{array}{lll}
 & (\mathbf{st}\ \{i \mapsto 1, s \mapsto 0\} \bullet S(2)) & \\
\xrightarrow{\top} & (\mathbf{st}\ \{i \mapsto 1, s \mapsto 0\} \bullet & \\
 & \quad (s \coloneqq s + i) \rightarrow (i \coloneqq i + 1) \rightarrow S(2)) & \mathit{Rules}\ (22), (23), (25) \\
\xrightarrow{\top} & (\mathbf{st}\ \{i \mapsto 1, s \mapsto 1\} \bullet (i \coloneqq i + 1) \rightarrow S(2)) & \mathit{Rules}\ (22), (27) \\
\xrightarrow{\top} & (\mathbf{st}\ \{i \mapsto 2, s \mapsto 1\} \bullet S(2)) & \mathit{Rules}\ (22), (27) \\
\xrightarrow{\top} & (\mathbf{st}\ \{i \mapsto 2, s \mapsto 1\} \bullet & \\
 & \quad (s \coloneqq s + i) \rightarrow (i \coloneqq i + 1) \rightarrow S(2)) & \mathit{Rules}\ (22), (23), (25) \\
\overset{\top}{\Longrightarrow} & (\mathbf{st}\ \{i \mapsto 3, s \mapsto 3\} \bullet S(2)) & \mathit{Rules}\ (22), (27)\ (\times 2) \\
\xrightarrow{\top} & (\mathbf{st}\ \{i \mapsto 3, s \mapsto 3\} \bullet x \coloneqq s \rightarrow \mathit{SKIP}) & \mathit{Rules}\ (22), (23), (25) \\
\xrightarrow{x \coloneqq 3} & (\mathbf{st}\ \{i \mapsto 3, s \mapsto 3\} \bullet \mathit{SKIP}) & \mathit{Rules}\ (22), (26)
\end{array}$$

It is a series of unobservable steps (in $\top$) until the final observable transition which updates x to 3. No more transitions are possible. (We have used $\overset{\top}{\Longrightarrow}$ to indicate a sequence of more than one $\xrightarrow{\top}$ transitions.) The steps in $\top$ may be interleaved with other processes operating in parallel by Rule (8). The process avoids internal "divergence" by eventually updating a non-local variable.

4 Combining Synchronisation and State-Based Actions

We have so far given a relatively small and straightforward extension to CSP to allow state-based behaviour. However the language is limited in that one cannot combine guards, updates and events in one atomic action. To this end we generalise guards and updates to *specification commands* (Sect. 4.1), which allow arbitrary relationships between pre- and post- states, and allow specification commands to be paired with events (Sect. 4.2).

4.1 Specification Commands

A specification command is of the form $x\colon [R]$, where x is a set of variables and R is a two-state predicate. This construct is based on Morgan's specification command [9], except he uses x_0 and x for pre- and post-state variables, where we use x and x', respectively. The *frame* of the command is the set x, i.e., $\mathit{frame}(x\colon [R]) = x$, and it is the set of variables which may be modified by the specification command. R defines the relationship between the values of pre- and post variables. Variables in the post-state are primed versions, and only variables in the frame x may appear primed in R. For example, a specification command which increments i is written as $i\colon [i' = i + 1]$. A guard as described in the previous section is a special case of a specification command where x is empty and R (therefore) does not refer to the post-state, that is, $[g]$ is now an abbreviation for $\varnothing\colon [g]$, as in [9]. Similarly, an update $x \coloneqq E$ is an abbreviation for $x\colon [x' = E]$. The syntax of processes is extended to allow a specification command as a prefix, which, given these abbreviations, subsumes prefixing by guards and assignments.

$$P ::= (\mathbf{st}\ \sigma \bullet P) \mid (x\colon [R] \rightarrow P) \mid \ldots \tag{31}$$

where '...' includes the operators for CSP from (1). We require $\alpha(x\colon [R] \rightarrow P) = \alpha(P)$.

As in the previous extension, we define the set of state-based actions, SCmd, to contain all specification commands (and allow guard and update abbreviations to appear in transition labels). In keeping with the guard abbreviation, we define $\top$ as any command $\varnothing\colon [g]$ where $[\![g]\!] = \mathit{true}$. The following rule subsumes Rule (22).

$$(x\colon [R] \rightarrow P) \xrightarrow{x\colon [R]} P \tag{32}$$

Given a two-state predicate R and states σ and σ', the substitution of σ in the pre-state of R and σ' in the post-state of R is written $R[\sigma, \sigma']$. For instance, if $\sigma = \{i \mapsto 0\}$ and $\sigma' = \{i \mapsto 1\}$, then $(i' = i + 1)[\sigma, \sigma']$ is $(1 = 0 + 1)$. A specification command $x\colon [R]$ is satisfiable when there exists some state σ and values for the frame variables x such that R holds for the pre-state σ and post-state formed from σ updated with the new values for the variables in x.

$$\mathbf{sat}(x\colon [R]) \mathrel{\widehat{=}} (\exists\, \Sigma \bullet (\exists\, V : (x \rightarrow \mathit{Val}) \bullet R[\Sigma, \Sigma \oplus V]))$$

The rule for a specification command transition in a local state is given below.

$$\frac{P \xrightarrow{x\colon [R]} P' \quad y = x \cap \operatorname{dom} \sigma \quad z = x \setminus \operatorname{dom} \sigma \quad V \in (y \rightarrow \mathit{Val}) \qquad \sigma' = \sigma \oplus V \quad \mathbf{sat}(z\colon [R[\sigma, \sigma']])}{(\mathbf{st}\ \sigma \bullet P) \xrightarrow{z\colon [R[\sigma, \sigma']]} (\mathbf{st}\ \sigma' \bullet P')} \tag{33}$$

A process $(\mathbf{st}\ \sigma \bullet P)$ may take a transition labelled by a specification command $z\colon [R[\sigma, \sigma']]$ under the following conditions. P transitions in specification command $x\colon [R]$ to P'. Set y is the subset of variables of x that are in the local state σ; z is the remaining variables. Choose some new values for the variables in y and call this state V. Then the new state σ' is the same as σ with variables in y updated according to V. Finally, the specification command $z\colon [R[\sigma, \sigma']]$ must be satisfiable. Then the conclusion of the rule states that $(\mathbf{st}\ \sigma \bullet P)$ transitions to $(\mathbf{st}\ \sigma' \bullet P')$ with label $z\colon [R[\sigma, \sigma']]$, i.e., the visible behaviour is an update of non-local variables z such that R holds, after variables in the local state σ are replaced by their local values in the pre- and post-states. If z is empty and $[\![R[\sigma, \sigma']]\!]$, the label is $\varnothing\colon [\mathit{true}]$, i.e., $\top$.

For example, consider a specification command which swaps the values of two variables, i and j, if both are greater than 0, but blocks otherwise.

$$i, j\colon [R] \qquad \text{where } R \mathrel{\widehat{=}} i > 0 \wedge j > 0 \wedge i' = j \wedge j' = i$$

Consider the execution of $(i, j\colon [R] \rightarrow Q)$ in a context which maps i to 5 and j to 10. The guard is satisfied since both i and j are non-zero, and the result is to swap their values. The following transition is justified by Rules (33) and (32).

$$(\mathbf{st}\ \{i \mapsto 5, j \mapsto 10\} \bullet i, j\colon [R] \rightarrow Q) \xrightarrow{\top} (\mathbf{st}\ \{i \mapsto 10, j \mapsto 5\} \bullet Q)$$

The instantiations of the rule meta-variables are $y = \{i, j\}, z = \varnothing$, $V = \{i \mapsto 10, j \mapsto 5\} = \sigma'$. Substituting σ and σ' into R gives *true*, and hence $R[\sigma, \sigma']$ is trivially satisfiable. No other choice for V would give a valid transition, since the satisfiability constraint would not hold.

All rules in Figs. 1 and 2 remain valid. In particular, Rule (33) specialises to Rule (25) with the following instantiations: $x = y = z = \{\}$; hence $V = \{\}$ and $\sigma' = \sigma$. Rule (33) specialises to Rule (26) for $x \notin \operatorname{dom} \sigma$ by replacing the abbreviation $x \mathrel{:=} E$ by $x\colon [x' = E]$, with the following instantiations: $y = \varnothing, z = \{x\}$, and hence $V = \varnothing$ and $\sigma' = \sigma$. We assume that E does not contain any primed variables, and therefore $\mathbf{sat}(x\colon [x' = E[\sigma]])$ holds because it simplifies to $(\exists\, \Sigma \bullet (\exists\, w : \mathit{Val} \bullet w = E[\Sigma]))$ which is true by the one-point law. Rule (33) specialises to Rule (27) for $x \in \operatorname{dom} \sigma$, by replacing the abbreviation $x \mathrel{:=} E$ by $x\colon [x' = E]$, with the following instantiations: $y = \{x\}, z = \varnothing$, and hence, for some v, $V = \{x \mapsto v\}$ and $\sigma' = \sigma \oplus \{x \mapsto v\}$. We assume that E does not contain any primed variables, and therefore $z\colon [R[\sigma, \sigma']]$ is $\varnothing\colon [v = E[\sigma]]$.

4.2 Combining State and Synchronisations

To allow more generality, we now allow each action in the language to be a specification command/event pair, written (c, μ). This pair subsumes both specification command prefix and event prefix. If c is $\top$ we abbreviate (c, μ) to μ, and if μ is τ we abbreviate (c, μ) to c. We allow τ as an abbreviation for $(\top, \tau)$. We require

$$\alpha((c, \mu) \rightarrow P) = \alpha(P) \quad \text{where } \mu \in \mathit{Event} \Rightarrow \mu \in \alpha(P)$$

The intuition is that an action (c, a) can synchronise on an event a if the guard for c holds, and will update the variables in the frame according to c. For instance, to correct the problem with possible refusals of *enq* events associated with process (14), the bottom line can be written as

$$([q \neq \langle\rangle], \mathit{deq}!\mathit{head}(q)) \rightarrow (q \coloneqq \mathit{tail}(q)) \rightarrow Q$$

This ensures that the synchronisation on channel *deq* will occur only when q is nonempty, without precluding the choice to enqueue a value.

For syntactic convenience in specifying a sequence of state-based actions to be executed atomically, we allow action pairs to be composed. Such a composition is well-formed only if at most one of the commands has an event a. That is, two events cannot be composed. We use the symbol '$\circ$' to denote composition of action pairs, and overload it to also denote relational composition of specification commands.

$$(c_1, \tau) \circ (c_2, \tau) = (c_1 \circ c_2, \tau) \tag{34}$$

$$(c_1, a) \circ (c_2, \tau) = (c_1 \circ c_2, a) = (c_1, \tau) \circ (c_2, a) \tag{35}$$

The relational composition of two specification commands, $c_1 \circ c_2$, which have the same frame x, is defined by matching the post-state of c_1 to the pre-state of c_2 via intermediate variables x''.

$$x\colon [R_1] \circ x\colon [R_2] = x\colon [\exists\, x'' \bullet R_1[\frac{x''}{x'}] \wedge R_2[\frac{x''}{x}]] \tag{36}$$

The expression $R_1[\frac{x''}{x'}]$ is R_1 with a syntactic replacement of variables x' with x''. Note that this is a different type of substitution to that involving states. For the purposes of defining relational composition when the frames do not match, their frames may be widened according to the rule below.

$$x\colon [R] = x \cup y\colon [R \wedge y' = y] \qquad \text{for } x \cap y = \varnothing \tag{37}$$

For example, the dequeuing of an element can be merged with the test of non-emptiness and an event on channel *deq* into a single atomic action as follows.

$$\begin{array}{ll}
\quad ([q \neq \langle\rangle], \mathit{deq}!\mathit{head}(q)) \circ (q \coloneqq \mathit{tail}(q), \tau) & \\
= ([q \neq \langle\rangle] \circ q \coloneqq \mathit{tail}(q), \mathit{deq}!\mathit{head}(q)) & \text{from (35)} \\
= (q\colon [q \neq \langle\rangle \wedge q' = q] \circ q\colon [q' = \mathit{tail}(q)], \mathit{deq}!\mathit{head}(q)) & \text{from (37)} \\
= (q\colon [\exists\, q'' \bullet q \neq \langle\rangle \wedge q'' = q \wedge q' = \mathit{tail}(q'')], \mathit{deq}!\mathit{head}(q)) & \text{from (36)} \\
= (q\colon [q \neq \langle\rangle \wedge q' = \mathit{tail}(q)], \mathit{deq}!\mathit{head}(q)) & \text{one-point rule}
\end{array}$$

In addition to allowing paired actions, we now require transition labels to be pairs of commands and events. As above, a transition with command c and event μ is written $P \xrightarrow{c,\mu} P'$. Both fields are mandatory for every transition, however, if the command is $\top$ or the event is τ, we omit them, with the minimum label being τ. As an example of a non-trivial label, by the above calculation:

$$[q \neq \langle\rangle] \circ \mathit{deq}!\mathit{head}(q) \circ q \coloneqq \mathit{tail}(q) \rightarrow P \xrightarrow{q\colon [q \neq \langle\rangle \wedge q' = \mathit{tail}(q)], \mathit{deq}!\mathit{head}(q)} P$$

$$((c,\mu) \rightarrow P) \xrightarrow{c,\mu} P \quad (38) \qquad \frac{P \xrightarrow{c,\mu} P' \quad \mu \neq \checkmark}{P\,;\,Q \xrightarrow{c,\mu} P'\,;\,Q} \quad \frac{P \xrightarrow{\checkmark} P'}{P\,;\,Q \xrightarrow{\tau} Q} \quad (40)$$

$$\frac{P \xrightarrow{c,a} P' \quad a \notin \alpha(Q)}{P \parallel Q \xrightarrow{c,a} P' \parallel Q} \quad (39) \qquad \frac{P \xrightarrow{c,a} P' \quad a \in A}{P \backslash A \xrightarrow{c,\tau} P' \backslash A} \quad \frac{P \xrightarrow{c,\mu} P' \quad \mu \notin A}{P \backslash A \xrightarrow{c,\mu} P' \backslash A} \quad (41)$$

and similarly for Q.

$$\frac{P \xrightarrow{\tau} P'}{P \parallel\!\!\!\parallel Q \xrightarrow{\tau} P' \parallel\!\!\!\parallel Q} \qquad \frac{P \xrightarrow{c,\mu} P' \quad (c,\mu) \neq (\top,\tau)}{P \parallel\!\!\!\parallel Q \xrightarrow{c,\mu} P'} \qquad (42)$$

and similarly for Q.

$$\frac{P \xrightarrow{x_1\colon [R_1],a} P' \quad Q \xrightarrow{x_2\colon [R_2],a} Q' \quad x_1 \cap x_2 = \varnothing}{P \parallel Q \xrightarrow{(x_1 \cup x_2)\colon [R_1 \wedge R_2],a} P' \parallel Q'} \qquad (43)$$

$$\frac{\begin{array}{c} P \xrightarrow{x\colon [R],\mu} P' \quad y = x \cap \operatorname{dom}\sigma \quad z = x \setminus \operatorname{dom}\sigma \quad V \in (y \rightarrow \mathit{Val}) \\ \sigma' = \sigma \oplus V \quad \mathbf{sat}(z\colon [R[\sigma,\sigma']]) \end{array}}{(\mathbf{st}\ \sigma \bullet P) \xrightarrow{z:[R[\sigma,\sigma']],\mu[\sigma]} (\mathbf{st}\ \sigma' \bullet P')} \qquad (44)$$

Fig. 3. Rules for specification command/event pairs

We must now define the rules for paired specification command/event transition labels. This is just a matter of combining the rules from Figs. 1 and 2; the result is given in Fig. 3. Rules (6) and (9) do not change. Rule (44) is a combination of Rules (24) and (33). Note that when c is $\top$, the rules collapse to those for CSP given in Fig. 1, and when μ is τ and c is a guard or update, the rules collapse to those in Fig. 2. Rule (43) allows two specification commands to be conjoined if their frames are disjoint. Recall that to be well-formed a specification command cannot alter variables outside its frame. This allows concurrent (and atomic) updates of distinct variables in separate threads.

4.3 Example

Recall the processes S, R and Sys from (4) and Qu from (14). So that CSP$_\sigma$ may be compared more readily with CSP, these processes have been defined using communication over channels. We now rewrite these processes using state variables to communicate data. We assume that Sys interacts with its environment by receiving information on channel ci and passing information on co. Firstly, we assume x is the only non-local variable that process *Produce* in S visibly alters, and therefore *Produce* does not need to be parameterised. An implementation of *Produce* may be a process which reads some value from the environment on channel ci then finds the sum to that number, i.e.,

$$\mathit{Produce} \mathrel{\widehat{=}} ci?X \rightarrow \mathit{Sum}(X)$$

where $\mathit{Sum}(X)$ is defined in (15). Process $\mathit{Consume}(y)$ used in R in (4) may perform some actions that manipulate non-local variable y (possibly involving local variables), and then output the result to the environment on channel co. For the purposes of this example we define $\mathit{Consume}$ minimally as a process which outputs y on channel co, i.e., $\mathit{Consume}(y) \mathrel{\widehat{=}} co!y \rightarrow \mathit{SKIP}$.

The next transformation we make is to replace communication with the Qu thread by direct operations on the variable q. In addition, we introduce the scope of x and y to contain S and R, respectively. These states are external to S and R because these processes are recursively defined, and would otherwise result in a series of redundant nested states.

$$\begin{aligned} S &\mathrel{\widehat{=}} \mathit{Produce} \mathbin{;} (q \coloneqq q \frown \langle x \rangle \rightarrow S) \\ R &\mathrel{\widehat{=}} y, q \colon [q = \langle y \rangle \frown q'] \rightarrow (\mathit{Consume}(y) \mathbin{;} R) \\ \mathit{Sys} &\mathrel{\widehat{=}} (\mathbf{st}\ \{q \mapsto \langle\rangle\} \bullet (\mathbf{st}\ \{x \mapsto 0\} \bullet S) \parallel (\mathbf{st}\ \{y \mapsto 0\} \bullet R)) \end{aligned} \tag{45}$$

Process S involves three layers of state. Within $\mathit{Produce}$ are the local variables of Sum, i and s, which are not visible. Variable x is external to Sum and $\mathit{Produce}$ but local to S; while q is external to S but local to Sys. The variables i and s are *local* variables, while q is a *shared* variable, however, the semantics makes no distinction.

A trace of Sys will appear as a sequence of communications on channels ci and co, interspersed with steps in τ. We build such a trace below. For space reasons we write a process $(\mathbf{st}\ \{x \mapsto 0\} \bullet P)$ as $P^{x=0}$, and write $P \stackrel{s}{\Longrightarrow} Q$ to represent a transition formed from multiple steps, in which the only externally visible transitions are those in s. Trace (46) represents possible executions of processes S and R. Process S receives the value 2 on channel ci from the environment, the sum to 2 is evaluated and stored in x, and q is updated to contain the value in x. Process R takes y from the queue and outputs it on channel co. Trace (47) is a trace formed from S and R in parallel, observed externally to the local variables x and y. Note that the values of those variables change within the state, and that references to them are replaced in the transition labels. Trace (48) represents a trace of the process Sys viewed externally; only communication with the environment is visible, with the updates to q reflected in the local state.

$$S \stackrel{ci.2/x \coloneqq 3/q \coloneqq q \frown \langle x \rangle}{\Longrightarrow} S \qquad R \stackrel{y,q\colon [q = \langle y \rangle \frown q']/co.y}{\longrightarrow} R \tag{46}$$

$$\begin{aligned} (S^{x=0} \parallel R^{y=0}) &\stackrel{ci.2/q \coloneqq q \frown \langle 3 \rangle}{\Longrightarrow} (S^{x=3} \parallel R^{y=0}) \\ &\stackrel{q\colon [q \neq \langle\rangle \wedge q' = tail(q)]/co.3}{\Longrightarrow} (S^{x=3} \parallel R^{y=3}) \end{aligned} \tag{47}$$

$$(S^{x=0} \parallel R^{y=0})^{q=\langle\rangle} \stackrel{ci.2}{\Longrightarrow} (S^{x=3} \parallel R^{y=0})^{q=\langle 3 \rangle} \stackrel{co.3}{\Longrightarrow} (S^{x=3} \parallel R^{y=3})^{q=\langle\rangle} \tag{48}$$

5 Related Work

5.1 Comparison with CSP

We first note that $(P \parallel P) = P$ does not hold if P accesses non-local variables, even in the absence of internal choice (demonic nondeterminism). Intuitively, this is as expected because successive updates of the same variable could result in a different final value. Technically, this is because specification commands are interleaved by Rule (39) (and an *SCmd* does not appear in the alphabet of any process), unless accompanied by a synchronisation event (Rule (43)).

The language extensions and semantics we have introduced in this paper provide a notational convenience for specifying state-based process behaviour. However, any process written in CSP$_\sigma$ can be transformed into a process in CSP if state values are kept as parameters to processes and channels are used to exchange information instead of shared variables. To transform a process in CSP$_\sigma$ to CSP, any accesses of a variable that is shared among multiple parallel processes must become encapsulated by a separate process. For instance, the queue is represented by a variable in (45), while in (4) it must be represented by a separate process, in this case, $Qu(\langle\rangle)$.

The CSP process in (2) gives an encapsulated data type Qu, which in some contexts is desirable, but can become cumbersome in others. Consider an extension of the CSP process Sys (4) that contains n instances of process R. We add a parameter to R, indicated by a subscript $i \in 1..n$, and must distinguish the deq channels so that pairs R_i and R_j do not synchronise with each other, but only with Qu. We also parameterise $Consume$ in a similar manner, so that interactions with the environment do not need to synchronise between R_is. The $Qu(q)$ process must also be updated to listen on multiple channels. The relevant definitions are given below.

$$\begin{array}{rl} R_i \mathrel{\widehat=} & deq_i?y \rightarrow (Consume_i(y)\,;\,R_i) \\ Qu(\langle y\rangle \frown q) \mathrel{\widehat=} & \begin{array}{l} enq?x \rightarrow Qu(\langle y\rangle \frown q \frown \langle x\rangle) \\ \mathbin{[\!]}\ (\mathbin{[\!]}_i\, deq_i!y \rightarrow Qu(q)) \end{array} \end{array} \tag{49}$$

We have used a generalised external choice to specify that the deq channel can output (synchronise) for any $i \in 1..n$.

Consider a further modification such that each R_i consumes two *successive* elements of the queue. However $R_i \mathrel{\widehat=} deq_i?y \rightarrow deq_i?z \rightarrow (Consume_i(y,z)\,;R_i)$. is insufficient as it does not ensure y and z were successively enqueued, since other deq_j events may interleave between. One solution is to define another event, e.g., $deq2_i?(y,z)$ which will atomically dequeue two elements. The definition is trivial, but requires a third process definition to be added to (49) (we assume that capability to dequeue a single element is still required). The Sys process is also updated to hide the new event and include n instances of R, for which we assume a generalised parallel composition operator.

$$\begin{array}{l} Queue2(\langle\rangle) \mathrel{\widehat=} \ldots \qquad Queue2(\langle y\rangle) \mathrel{\widehat=} \ldots \\ Queue2(\langle y,z\rangle \frown q) \mathrel{\widehat=} \begin{array}{l} \quad enq?x \rightarrow \ldots \\ \mathbin{[\!]}\ (\mathbin{[\!]}_i\, deq_i!y \rightarrow Queue2(\langle z\rangle \frown q)) \\ \mathbin{[\!]}\ (\mathbin{[\!]}_i\, deq2_i!(y,z) \rightarrow Queue2(q)) \end{array} \\ S \mathrel{\widehat=} (Produce(x)\,;\,(enq!x \rightarrow S)) \\ R_i \mathrel{\widehat=} deq2_i?(y,z) \rightarrow (Consume_i(y,z)\,;\,R_i) \\ Sys \mathrel{\widehat=} (S \parallel (\parallel_i R_i) \parallel Queue2) \backslash (enq.Val \cup (\bigcup_i deq2_i.V)) \end{array} \tag{50}$$

The approach of adding new operations does not scale for complex data types which require arbitrary, atomic combinations of operations. It is more convenient to specify this behaviour directly in a state-based style. Recall from (45) that there is no need for a queue process, as the queue operations may be merged into S and R directly. Similarly, we are able to define the operation of removing two adjacent elements in one atomic action. The *Sys* process, with n instances of R and double-dequeue is shown below.

$$\begin{aligned} S &\mathrel{\widehat=} \mathit{Produce}\ ; ((q \coloneqq q \frown \langle x \rangle) \rightarrow S) \\ R_i &\mathrel{\widehat=} y, z, q \colon [q = \langle y, z \rangle \frown q'] \rightarrow (\mathit{Consume}_i(y, z)\ ; R_i) \\ \mathit{Sys} &\mathrel{\widehat=} (\mathbf{st}\ \{q \mapsto \langle\rangle\} \bullet S^{x=0} \parallel (\|_i\ {R_i}^{y=0, z=0})) \end{aligned} \tag{51}$$

Note that there are n declarations of the local variables y and z, but we do not need to subscript them to distinguish the different instances. All references to y or z in R_j will properly reference the correct version.

Given that a new process *Queue*2 does not need to be defined and internal messages do not need to be hidden, (51) is a more compact specification than that given by (50). It is also flexible enough if other operations on the queue are required. In CSP, all shared data must have an opaque type, however, as demonstrated above, in CSP$_\sigma$ shared data may have their types exposed. Of course, it is a separate question as to whether process algebras such as CSP should be used for defining programs like *Sys*. Hoare gives laws for assignment and state variables, but concludes that the laws are not mathematically convenient, and that "... there are adequate grounds for introducing the assignable program variable as a new primitive concept" [7, Sect. 5.5.3]. We intend CSP$_\sigma$ to be in keeping with the spirit of this statement, and the specification style of CSP in general.

5.2 Other Work

CSP has been integrated with state-based languages, for instance, with Z by Woodcock & Cavalcanti (Circus) [14], with Object-Z by Smith and Fischer & Wehrheim (CSP-OZ) [13,5], with Action Systems by Butler [2], and with B by Butler & Leuschel [1]. In comparison with these approaches to combining state-based specification with CSP, we have taken a "lightweight" approach, integrating only a single construct for defining state manipulation, and with little change to the underlying syntax and semantics of CSP. Of course, the addition of state tests and updates does not provide the same richness of specification as afforded by a full incorporation of Z, etc., but may provide a useful stepping stone between event- and state-based specifications.

Plotkin's seminal paper on operational semantics [10] defines transition rules for imperative languages with state. There are also many other examples of such semantics in the literature, in particular, the semantics of Hoare and He Jifeng [8], and the semantics for the programming language Occam [6]. Our approach is different in that the state is treated as part of the process, and guards and updates are treated as labels to the transition relation. This allows state accesses to be (perhaps partially) hidden by an outer context which defines the values of the local state. The traditional operational semantics approach defines the transition relation on program/state pairs, and the state is updated in the rule for each construct (e.g., update). This approach does not so easily support the hierarchical construction of the state as in our approach, with local variables in the traditional style being captured as global variables with syntactic restrictions. In the approach adopted here, by treating state access as transition labels, the state-based reasoning is 'quarantined' to a single, general rule (Rule (44)), allowing the construct rules, e.g., Rule (22) and Rule (32), to be defined concisely, and without explicit reference to a particular state. This extends to more complex constructs, for instance, the transition rules for *conditional* can be given without reference to state (see Rule (52)).

$$(\mathbf{if}\ b\ \mathbf{then}\ P\ \mathbf{else}\ Q) \xrightarrow{[b]} P \qquad (\mathbf{if}\ b\ \mathbf{then}\ P\ \mathbf{else}\ Q) \xrightarrow{[\neg b]} Q \tag{52}$$

6 Conclusions

In this paper we have given an extension to the CSP language which allows state-based constructs to be integrated with inter-process synchronisation and other CSP constructs. The extension is given an operational semantics, defined so that it is also an extension of the CSP operational semantics: all existing transitions are preserved. The approach taken to defining the transition rules is novel in that the state is maintained as part of the process, instead of a meta-level construct in the rules, and that, therefore, transitions are labelled by specification commands. This enables a more compact presentation and naturally leads to hierarchical definition of states.

The work was motivated when developing a semantics for Behavior Trees [3], a notation used for capturing natural language requirements of large systems. Such requirement documents often mix styles and levels of specification, e.g., a system of systems will specify both the interactions and the internal operations of the processes involved. In future work we will extend CSP$_\sigma$ to handle the full range of Behavior Tree constructs. In preparation for developing code from CSP$_\sigma$, we will define a trace-based semantics for refinement of CSP$_\sigma$.

Because the extension builds on existing constructs and semantics, the state-based rules should fit with existing tool support for CSP, such as FDR [4]. However, state-space explosion will become an issue with unrestricted types, and evaluation strategies must be devised to avoid efficiency issues with checking satisfiability in Rule (44).

Acknowledgments. The authors thank three anonymous reviewers for helpful comments on this paper. Ian Hayes would like to acknowledge his Visiting Professorship at, and the hospitality of, the University of Newcastle (UK), and support by the EPSRC-funded Trustworthy Ambient Systems (TrAmS) Platform Project.

References

1. Butler, M.J., Leuschel, M.: Combining CSP and B for Specification and Property Verification. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 221–236. Springer, Heidelberg (2005)
2. Butler, M.J.: A CSP Approach to Action Systems. PhD thesis, Computing Laboratory, Oxford Univ. (1992)
3. Dromey, R.G.: Formalizing the transition from requirements to design. In: Jifeng, H., Liu, Z. (eds.) Mathematical Frameworks for Component Software: Models for Analysis and Synthesis, River Edge, NJ, USA. Component-Based Development, pp. 156–187. World Scientific Publishing Co., Inc., Singapore (2006)
4. FDR2 user manual (2005), `http://www.fsel.com/fdr2_manual.html`
5. Fischer, C., Wehrheim, H.: Model-Checking CSP-OZ Specifications with FDR.. In: Araki, K., Galloway, A., Taguchi, K. (eds.) Integrated Formal Methods, 1st International Conference, Proceedings, pp. 315–334. Springer, Heidelberg (1999)
6. Gurevich, Y., Moss, L.S.: Algebraic operational semantics and Occam. In: Börger, E., Kleine Büning, H., Richter, M.M. (eds.) CSL 1989. LNCS, vol. 440, pp. 176–192. Springer, Heidelberg (1990)
7. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Inc., Englewood Cliffs (1985)
8. Hoare, C.A.R., Jifeng, H.: Unifying Theories of Programming. Prentice-Hall, Englewood Cliffs (1998)
9. Morgan, C.: Programming from Specifications, 2nd edn. Prentice-Hall, Englewood Cliffs (1994)
10. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebr. Program. 60-61, 17–139 (2004)
11. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall, Englewood Cliffs (1998)
12. Schneider, S.: Concurrent and Real-time Systems: The CSP Approach. Wiley, Chichester (2000)
13. Smith, G.: A semantic integration of Object-Z and CSP for the specification of concurrent systems. In: Fitzgerald, J.S., Jones, C.B., Lucas, P. (eds.) FME 1997. LNCS, vol. 1313, pp. 62–81. Springer, Heidelberg (1997)
14. Woodcock, J.C.P., Cavalcanti, A.L.C.: The semantics of circus. In: Bert, D., Bowen, J.P., Henson, M.C., Robinson, K. (eds.) B 2002 and ZB 2002. LNCS, vol. 2272, pp. 184–203. Springer, Heidelberg (2002)

Predicate Abstraction in a Program Logic Calculus

Benjamin Weiß

Institute for Theoretical Computer Science
University of Karlsruhe, D-76128 Karlsruhe, Germany
bweiss@ira.uka.de

Abstract. Predicate abstraction is a form of abstract interpretation where the abstract domain is constructed from a finite set of predicates over the variables of the program. This paper explores a way to integrate predicate abstraction into a calculus for deductive program verification, where it allows to infer loop invariants automatically that would otherwise have to be given interactively. The approach has been implemented as a part of the KeY verification system.

1 Introduction

Deductive verification of imperative programs typically requires hand-crafted *loop invariants*, i.e., assertions about the program states which can possibly occur at the beginning of each iteration of a loop. Finding sufficiently strong loop invariants can be difficult, and today this is often one of only a few human interactions necessary in an otherwise heavily automated verification environment.

On the other hand, there are methods which can automatically determine loop invariants. Leaving aside testing-based approaches like Daikon [9], such methods are predominantly based on *abstract interpretation* [6], a theoretical framework for static program analysis which can roughly be described as symbolic execution of the program, using an abstract (i.e., approximative) domain for the variable values, together with fixed-point iteration.

Predicate abstraction [11] is a variant of abstract interpretation where the abstract domain is constructed from a finite set of predicates over the variables of the program. Here, the symbolic execution is itself done in a precise fashion, and the necessary approximation is performed in between by explicit abstraction steps, in which an automated theorem prover is used to determine a valid boolean combination of the predicates. Compared with other forms of abstract interpretation, a fundamental disadvantage of predicate abstraction is that it is limited to *finite* abstract domains. On the other hand, an advantage is that its abstract domain can be flexibly adapted by simply changing the set of predicates. In the same vein, predicate abstraction can quite easily support complex, quantified invariants [10]. It can be extended with an iterative refinement process that automatically adapts the domain to the particular problem [5].

This paper presents an approach for integrating predicate abstraction into a deductive program verification calculus. This allows to infer loop invariants within this calculus, on demand and as an integral part of constructing the overall correctness proof.

Outline. Sect. 2 gives an overview of relevant related work. Necessary background on the underlying program logic and calculus is provided in Sect. 3. A high level explanation of the approach follows in Sect. 4. In Sect. 5, new calculus rules are introduced, and how these rules are to be used is described in Sect. 6. Sect. 7 gives details on the predicate abstraction scheme. The overall method is further illustrated with the help of an example in Sect. 8, and practical experience with an implementation is reported in Sect. 9. Finally, Sect. 10 contains conclusions and future work.

2 Related Work

This paper draws much inspiration from Flanagan and Qadeer's approach for using predicate abstraction in program verification [10]. Both in their approach and in ours, a set of predicates is associated with each loop in a program, and used to abstract specifically at loop entry points. Quantified loop invariants are supported by allowing the loop predicates to contain free variables which are later quantified over. The main difference is that in our setting, the inference is done within a logical calculus, the same that is used for the verification itself. This also distinguishes our technique from the one used in the Boogie verifier [2], where a separate abstract interpretation component is used to infer needed loop invariants, leading to a duplication of knowledge between the verifier and the abstract interpreter.

Several related approaches in striving for a closer integration between deductive verification and static analysis based invariant inference exist. In the "loop invariants on demand" technique [14], first-order verification conditions are generated from programs, which include placeholder predicates for the loop invariants. These are then passed to a first-order theorem prover. When an invariant is necessary for a sub-proof, the prover tries to infer it by repeatedly invoking an abstract interpreter with successively more precise abstract domains. Still, the verification condition generator, theorem prover, and abstract interpreter, are all separate components. In [15], parts of the invariant generation are moved inside the theorem prover, with the verification condition generation remaining separated. In our approach, all three tasks—especially generation of verification conditions and generation of invariants, which are closely related as they both deal with programs—can be performed within one program logic theorem prover. Logical interpretation [19] goes the other way round by embedding theorem proving techniques in an abstract interpretation framework.

The results presented in this paper are based on earlier work reported in [18]. Compared to [18], there are significant improvements: no unsoundness issues remain; the integration of invariant inference into the calculus is more natural, as proofs are no longer necessarily tree-shaped; and the transformation of state updates into formulas is now lazy instead of eager, which improves performance.

3 Program Logic

The verification framework used in this paper is *dynamic logic with non-rigid functions (DL)* [4,3], a generalisation of Hoare logic [12]. DL extends first-order logic by modal operators [p], where p can be any legal sequence of statements in some programming language. Additionally, it features modal operators $\{u\}$, where u is a so-called *update* [17], representing a state change in a language-independent, logical way. A core DL for a minimalist object-oriented language is formally defined in [4], and a full-blown version for Java in [3].

Formulas are evaluated in program states, which are first-order structures. The formula $[\mathtt{p}]\psi$ holds in a state if all states reachable by executing p in this state satisfy ψ. Similarly, $\{u\}\psi$ holds in a state if ψ holds in the state produced by the update u. A formula is *valid* if it holds in all states. A typical program verification task is to prove the validity of a formula $\varphi \rightarrow [\mathtt{p}]\psi$, which is equivalent to the Hoare triple $\{\varphi\}\mathtt{p}\{\psi\}$. Object attributes are represented as non-rigid function symbols, i.e., symbols whose interpretation may be changed by programs.

The validity of DL formulas can be proven using a sequent calculus. A *sequent* is a construct $\Gamma \vdash \Delta$, where Γ and Δ are finite sets of formulas, and whose semantics is the same as that of $\bigwedge \Gamma \rightarrow \bigvee \Delta$. A sequent calculus *rule* deduces the validity of a sequent (the rule's *conclusion*) from the validity of one or more other sequents (the rule's *premises*). In order to prove the validity of a sequent, one constructs a *proof tree*: its root is the original sequent itself, and in each step, it is extended by applying a rule to one of its leaves (called *goals*). Applying a rule means matching its conclusion to the goal, and adding its premises as children of the goal. If a proof goal is obviously valid (e.g., $\Gamma \vdash$ *true*), it is *closed.* If all goals of a proof tree are closed, this means that the root sequent is valid as well.

Formulas with programs in them may be handled by rules which operate on the *active statement*, i.e., the first basic command in the modal operator, and stepwise shorten the program until only a first-order problem remains. Intuitively, this process can be understood as *symbolic execution*: the program is "executed", but with symbolic instead of concrete values for its variables. It is similar to the *verification condition generation* in related verification approaches, but differs in that it is intertwined with other forms of reasoning, in particular first-order reasoning and arithmetic simplification, within the same calculus.

Such symbolic execution rules formalise the semantics of the underlying programming language. In the following, we take a look at rules for the three elementary programming constructs of assignments, conditional statements, and loops, in a Java-like language. The basic assignment rule is

$$\mathsf{assign}\ \frac{\Gamma \ \vdash \ \{u; \mathtt{x} := \mathtt{se}\}[\omega]\psi, \ \Delta}{\Gamma \ \vdash \ \{u\}[\mathtt{x\ =\ se;}\ \omega]\psi, \ \Delta}$$

where Γ and Δ are sets of formulas; u is an update; se is a "simple expression", i.e., an expression without side effects; ω is the rest of the program after the assignment; and ψ is a formula. The rule simply transforms the *program assignment* `x = se;` into an equivalent *update* $\mathtt{x} := \mathtt{se}$. The update $u; \mathtt{x} := \mathtt{se}$ is the sequential composition of the updates u and $\mathtt{x} := \mathtt{se}$. Parallel composition of updates is also possible; for example, $(\mathtt{x} := 1 \| \mathtt{y} := \mathtt{x})$ sets `x` to 1 and `y` to the value of `x` simultaneously. Finally, the update language allows quantified updates such as $(\mathit{for}\ x; \mathtt{a}[x] := 0)$, which sets all elements of the array `a` to 0 in parallel.

The assign rule reduces assignments to updates. In the course of symbolic execution, a composite update accumulates in this way in front of the modal operator. This update can be simplified aggressively using update rewriting rules [17], which for simplicity we use as a monolithic rule simplifyUpdate here. Once the program has been dealt with completely, the final update can be applied to the postcondition as a substitution (also by simplifyUpdate). As an example, consider the following unclosed proof tree (with the root at the bottom):

$$\begin{array}{c|c}
 & \vdash\ \mathit{if}(\mathtt{a} \doteq \mathtt{b})\mathit{then}(2)\mathit{else}(1) \doteq 1 \\ \hline
(\mathsf{simplifyUpdate}) & \vdash\ \{\mathtt{a.f} := 1; \mathtt{b.f} := 2\}(\mathtt{a.f} \doteq 1) \\ \hline
(\mathsf{assign}) & \vdash\ \{\mathtt{a.f} := 1;\}[\mathtt{b.f\ =\ 2;}]\mathtt{a.f} \doteq 1 \\ \hline
(\mathsf{simplifyUpdate}) & \vdash\ \{\mathtt{a.f} := 0; \mathtt{a.f} := 1;\}[\mathtt{b.f\ =\ 2;}]\mathtt{a.f} \doteq 1 \\ \hline
(\mathsf{assign}) & \vdash\ \{\mathtt{a.f} := 0\}[\mathtt{a.f\ =\ 1;\ b.f\ =\ 2;}]\mathtt{a.f} \doteq 1 \\ \hline
(\mathsf{assign}) & \vdash\ [\mathtt{a.f\ =\ 0;\ a.f\ =\ 1;\ b.f\ =\ 2;}]\mathtt{a.f} \doteq 1
\end{array}$$

Terms like `f(a)`, where `f` is a non-rigid function symbol, are written as `a.f` in order to resemble the usual object attribute access notation. One after the other, the three assignments are turned into updates. Since the first is overridden by the second, it can be simplified away. Finally, the update is applied to the postcondition $\mathtt{a.f} \doteq 1$ (expressing equality of `a.f` and 1). This last step creates a syntactical case distinction on whether `a` and `b` refer to the same object. Delaying and sometimes avoiding such *aliasing* related case distinctions is the primary motivation for handling assignments via updates in this way.

Conditional statements are symbolically executed by branching the proof on whether the guard is true or false, and loops by unwinding them:

$$\mathsf{ifElse}\ \frac{\begin{array}{ll}\Gamma,\ \{u\}\mathtt{se} \doteq \mathtt{true}\ \vdash\ \{u\}[\mathtt{p}\ \omega]\psi,\ \Delta & \text{(then branch)} \\ \Gamma,\ \{u\}\mathtt{se} \doteq \mathtt{false}\ \vdash\ \{u\}[\mathtt{q}\ \omega]\psi,\ \Delta & \text{(else branch)}\end{array}}{\Gamma\ \vdash\ \{u\}[\mathtt{if(se)\ p\ else\ q}\ \omega]\psi,\ \Delta}$$

$$\mathsf{loopUnwind}\ \frac{\Gamma\ \vdash\ \{u\}[\mathtt{if(e)\{p\ while(e)\ p\}}\ \omega]\psi,\ \Delta}{\Gamma\ \vdash\ \{u\}[\mathtt{while(e)\ p}\ \omega]\psi,\ \Delta}$$

Using loopUnwind is sufficient only for loops which terminate after a fixed, statically known number of iterations. General loops can be handled with loopInvariant (both loop rules are shown in a simplified form which assumes that the loop body does not terminate abruptly, e.g by throwing an exception):

$$\mathsf{loopInvariant}\ \frac{\begin{array}{ll}\Gamma\ \vdash\ \{u\}\mathit{Inv},\ \Delta & \text{(initially valid)} \\ \mathit{Inv},\ \mathtt{se} \doteq \mathtt{true}\ \vdash\ [\mathtt{p}]\mathit{Inv} & \text{(preserved by body)} \\ \mathit{Inv},\ \mathtt{se} \doteq \mathtt{false}\ \vdash\ [\omega]\psi & \text{(use case)}\end{array}}{\Gamma\ \vdash\ \{u\}[\mathtt{while(se)\ p}\ \omega]\psi,\ \Delta}$$

Here, *Inv* is a loop invariant which has to be provided from the outside. The first two branches ensure that *Inv* is indeed an invariant, i.e., that it holds both when initially encountering the loop and after an arbitrary number of loop iterations. In the third branch, symbolic execution continues behind the loop.

4 Approach

A program logic calculus like the one introduced in the previous section bears many similarities to abstract interpretation style program analysis; both use symbolic execution to infer and check properties about programs. Unlike usual abstract interpretations, the deductive approach can, at least in principle, handle arbitrarily precise properties. This comes at the cost of sometimes needing human interaction for proving the resulting first-order problems, and at the cost of requiring manually specified loop invariants. This paper aims to address the latter issue by integrating abstract interpretation concepts into the deductive setting.

A difference between abstract interpretation and our calculus is in the treatment of control flow splits: the calculus handles them by branching the proof tree, where the created branches remain separated permanently. On the other hand, abstract interpretations typically use a "merge" operator to combine properties at junction points in the control flow graph. This corresponds to accumulating properties for every program point, instead of treating the execution paths separately. For loops, the infinite number of paths makes such an accumulation necessary; deductive verification "cheats" here by assuming to be given a loop invariant, which already is an accumulated description of all paths through the loop. We can overcome this difference rather straightforwardly by introducing a rule into the calculus which merges several proof branches into one.

With this change, loops can be treated by applying loopUnwind and ifElse, symbolically executing the body, and then merging the resulting sequent (where the loop entry is again the active statement) with the previous such sequent. For example, we might begin with a sequent $\mathtt{i} \doteq 0 \vdash [\mathtt{while(i<j)\ \ldots}]\psi$. After one iteration, we might arrive at $\mathtt{i} \doteq 0 \lor \mathtt{i} \doteq 1 \vdash [\mathtt{while(i<j)\ \ldots}]\psi$, reflecting the fact that after this iteration, `i` has been incremented by one. Every such iteration leads to a larger set of states possible for the loop entry point. In principle, we only have to repeat this iterative process until this set of states stabilises, i.e., until it is a fixed point of the process: once this happens, it covers all states which are possible for the loop entry on any execution path, or in other words, its representation as a formula then is a loop invariant.

In the terminology of abstract interpretation, this would correspond to a computation of the *static semantics*. Obviously, the infinite number of states means that for most loops, such a computation would not terminate. To change this, we need to introduce *approximation*. A form of approximation particularly suitable in our context is that of predicate abstraction [11,10]: We assume that for each loop we are given a finite set P of predicates (formulas). Then, the abstraction of a formula for the entry point of this loop is a boolean combination of elements of P which is implied by the original formula. That is, the abstraction retains the information from the formula which is expressible by the predicates in P, and approximates away everything else. Since there are only finitely many boolean combinations of the predicates, performing such an abstraction before each unwinding step ensures convergence after a finite number of iterations. The found invariant can then be used to apply loopInvariant.

With predicate abstraction, the predicates P associated with a loop form the building blocks for the invariants which can be found for that loop. Such predicates can either be specified manually—which is easier than having to specify whole, correct loop invariants—or be generated heuristically based on the particular program and specification to be verified.

5 Rules

In this section, we define new sequent calculus rules which extend a rule base like the one sketched in Sect. 3 with predicate abstraction based loop invariant inference as described in Sect. 4. The soundness proofs for these rules are omitted here for space reasons. First is a rule for merging execution paths at junction points in the control flow graph, called merge:

$$\mathsf{merge}\ \frac{\bigwedge(\Gamma_1 \cup \neg\Delta_1) \vee \dots \vee \bigwedge(\Gamma_n \cup \neg\Delta_n)\ \vdash\ \psi}{\Gamma_1\ \vdash\ \psi, \Delta_1 \qquad \dots \qquad \Gamma_n\ \vdash\ \psi, \Delta_n}$$

This rule is unusual in that it has several conclusions, or in other words, in that it is applied to several proof goals at once. To allow such rules means to generalise the structure of proofs from trees to directed acyclic graphs (DAGs) which are connected and rooted. Apart from that, merge is a rather simple rule operating on the propositional logic level. A typical application (to be read, intuitively, from bottom to top) is

$$(\mathsf{merge})\frac{\varphi_1 \vee \varphi_2 \vdash [\mathtt{while(e)\ p}]\psi}{\varphi_1 \vdash [\mathtt{while(e)\ p}]\psi \qquad \varphi_2 \vdash [\mathtt{while(e)\ p}]\psi}$$

The next rule is responsible for the predicate abstraction step:

$$\mathsf{predicateAbstraction}\ \frac{\alpha_P(\bigwedge(\Gamma \cup \neg\Delta))\ \vdash\ [\mathtt{while(e)\ p}\ \omega]\psi}{\Gamma\ \vdash\ [\mathtt{while(e)\ p}\ \omega]\psi, \Delta}$$

where P is the set of predicates associated with the loop `while(e)p`, and where α_P is a meta-operator which computes for any formula φ a predicate abstraction using P. This means that $\alpha_P(\varphi)$ is some boolean combination of the predicates in P such that $\varphi \rightarrow \alpha_P(\varphi)$ is valid. The details of computing $\alpha_P(\varphi)$ depend on the particular predicate abstraction scheme (Sect. 7); usually, this computation itself requires first-order reasoning modulo several theories.

Both above rules operate on sequents without updates in front of the modal operators containing the programs. Thus, we need a way to transform typical sequents $\varphi \vdash \{u\}[\mathtt{p}]\psi$ such that the update u is removed from a modality $[\mathtt{p}]$. This can be achieved with the shiftUpdate rule:

$$\mathsf{shiftUpdate}\ \frac{\{u'\}\Gamma,\ \mathit{Upd}\ \vdash\ [\mathtt{p}]\psi,\ \{u'\}\Delta}{\Gamma\ \vdash\ \{u\}[\mathtt{p}]\psi,\ \Delta}$$

where:

– $\mathit{targets}(u)$ is the set of all (non-rigid) function symbols f occurring as top level operators of the left hand side of an elementary update $(f(\bar{t}) := t)$ in u

- for each $f \in \mathit{targets}(u)$: f' is a fresh rigid function symbol with the same arity as f
- the update u' is the parallel composition of the updates $(\mathit{for}\,\bar{x}; f(\bar{x}) := f'(\bar{x}))$ for all such pairs (f, f'), where $\bar{x} = x_1, \ldots, x_n$, n being the arity of f and f'
- $\mathit{Upd} = \bigwedge_{f \in \mathit{targets}(u)} \forall \bar{y}; f(\bar{y}) \doteq \{u'\}\{u\}f(\bar{y})$

Intuitively, the update u' substitutes for each updated function symbol f a fresh symbol f' which represents the old, pre-update, instance of f. The formula *Upd* links the old instances with the current ones. The following proof tree is an example:

$$\dfrac{\dfrac{\begin{array}{c} f'(\mathsf{a}) \doteq 27, \\ \forall y; y\texttt{.f} \doteq \mathit{if}(y \doteq \texttt{b})\mathit{then}(42)\mathit{else}(f'(y)) \\ \vdash\ [\texttt{p}]\psi \end{array}}{\begin{array}{c} \{\mathit{for}\ x; x\texttt{.f} := f'(x)\}\texttt{a.f} \doteq 27, \\ \forall y; y\texttt{.f} \doteq \{\mathit{for}\ x; x\texttt{.f} := f'(x)\}\{\texttt{b.f} := 42\}y\texttt{.f} \\ \vdash\ [\texttt{p}]\psi \end{array}}\ (\mathsf{simplifyUpdate})}{\texttt{a.f} \doteq 27\ \vdash\ \{\texttt{b.f} := 42\}[\texttt{p}]\psi}\ (\mathsf{shiftUpdate})$$

Since the updates resulting from this application of shiftUpdate are attached to formulas without modalities, they can be simplified away immediately, leading to a sequent without updates at all. This example also shows the disadvantage of using shiftUpdate, which is that it indirectly introduces quantifications and case distinctions for the possible aliasing situations. Using updates, instead of handling assignments in the style of shiftUpdate right away, allows to delay these complications as long as possible.

Finally, we introduce an operation setBack, which is defined as "replace a goal by one of its dominators in the proof graph". This is not strictly expressible as a sequent calculus rule, but it preserves the overall meaning of the proof: if all goals are valid, then the root must be valid. It is useful for "cutting off" proof branches which do not contribute to the loop invariant of the current loop. Such irrelevant branches for example occur when the loop body may throw an uncaught exception; the execution paths where this happens never return to the loop entry, and thus do not affect the loop invariant. Another example is the loop termination branch which is created when applying loopUnwind and subsequently ifElse. Instead of considering these side branches in every iteration of symbolic execution, they can be reverted to the loop entry with setBack. This is exemplified by the proof graph below:

$$(\mathsf{loopUnwind,\ ifElse})\dfrac{\varphi_1 \vdash [\texttt{p; while(e) p}]\psi \qquad (\mathsf{setBack})\dfrac{\varphi \vdash [\texttt{while(e) p}]\psi}{\varphi_2 \vdash []\psi}}{\varphi \vdash [\texttt{while(e) p}]\psi}$$

Instead of continuing on the right branch, it is set back to the loop entry. Once the loop body p has been symbolically executed on the left branch, merge can be used to combine both branches.

6 Proof Search Strategy

Sect. 4 has sketched the overall idea for how to apply the rules defined in Sect. 5. In this section, we concretise this aspect by defining a corresponding *proof search strategy*, i.e., an algorithm which automatically chooses the next rule to apply to a given unclosed proof. Our strategy extends a strategy able to do regular symbolic execution and first-order reasoning with the capability to infer a loop invariant whenever an invariant-less loop is encountered during proof construction.

The strategy is defined semi-formally in Fig. 1. The first three functions are helpers for the main function *chooseRuleApplication*. This function returns a pair of a goal node and a rule, with the meaning that the returned rule should be applied to the returned goal. The presentation is a bit imprecise in this respect, because in general there may of course be multiple ways to apply a single rule to a particular goal. However, for the rules that matter here, the exact application focus is either unique or it is explained in the paragraphs below. We assume that the occurring sequents are of the form $(\Gamma \vdash \{u\}[\mathsf{p}]\psi, \Delta)$, where p is the only program occurring in the sequent.

We consider a symbolic execution state, as captured by a node of the proof graph, to be "in" a loop when that loop has previously been "entered" by applying loopUnwind but not yet "left" by applying loopInvariant. Accordingly, the *entryNode* function determines the node where a specific loop, passed as a parameter to the function, has last been entered. Function *innermostLoop* returns the loop that has last been entered but not yet left.

Function *waiting* tells whether the symbolic execution of the passed node should not be continued yet, because operations on other branches have to be performed first. This is the case if the active statement is a loop, and if from the entry node of that loop it is possible to reach in the graph open goals where the active statement is not that loop: in this case, we first want to continue symbolic execution of these other goals until they get back to the loop as active statement. Only then do we continue with all of them, by combining them with merge.

The main function *chooseRuleApplication* now works as follows. First, it picks an arbitrary open goal which is not waiting for other branches. Then, it checks whether the innermost loop that symbolic execution is "in" does not occur in the program contained in the modal operator anymore. If so, this indicates that the current branch will not return to the loop entry, for example because an exception has been thrown which is not caught within the loop body. The next step is then to revert it to the entry point of the innermost loop with setBack. Otherwise, the choice of the rule depends on whether the active statement is a loop or not. If not, the strategy chooses a regular applicable symbolic execution rule or a first-order rule (abbreviated as SE in Fig. 1).

If the active statement is a loop, and if an invariant is already known for this loop, the invariant is used to apply loopInvariant. If no invariant is known, special rules are applied in a fixed order. First after reaching the loop entry via regular symbolic execution, shiftUpdate is used to get rid of any update preceding the modal operator. Then, merge can be applied to merge the current proof branch with all other branches that have been waiting for it. The next step

```
—— Pseudocode ——
//returns the node where symbolic execution entered a loop
Node entryNode(node, loop)
    if(activeStatement(node) = loop)
        if(appliedRule(node) = loopUnwind) return node;
        else if(appliedRule(node) = loopInvariant) return none;
    return entryNode(firstParent(node), loop);

//returns the innermost loop which symbolic execution is in
Loop innermostLoop(node, leftLoops)
    if(activeStatement(node) is a loop)
        if(appliedRule(node) = loopUnwind and loop $\notin$ leftLoops)
            return loop;
        else if(appliedRule(node) = loopInvariant) leftLoops := leftLoops $\cup$ {loop};
    return innermostLoop(firstParent(node), leftLoops);

//tells whether a node has to wait for other merge parents
boolean waiting(node)
    if(activeStatement(node) is a loop)
        foreach(goal reachable from entryNode(node, loop))
            if(open(goal) and activeStatement(goal) $\neq$ loop) return true;
    return false;

//main: chooses a goal and a rule which should be applied to the goal
(Node, Rule) chooseRuleApplication()
    goal := any goal with open(goal) and not waiting(goal);
    if(not occursIn(innermostLoop(goal, $\emptyset$), goal)) rule := setBack;
    else if(activeStatement(goal) is a loop)
        if(knownInvariant(loop) $\neq$ none) rule := loopInvariant;
        else lastRule := appliedRule(firstParent(goal));
            if(lastRule = SE) rule := shiftUpdate;
            else if(lastRule = shiftUpdate) rule := merge;
            else if(lastRule = merge) rule := predicateAbstraction;
            else if(lastRule = predicateAbstraction)
                if(fixed point) rule := loopInvariant;
                else rule := loopUnwind;
    else rule := SE;
    return (goal, rule);
—— Pseudocode ——
```

Fig. 1. Proof search strategy for predicate abstraction

is to perform predicate abstraction. Finally, if the iterative unwinding process has reached a fixed point, i.e., if the current abstraction (as returned by α_P) is logically equivalent to the previous abstraction for this loop, then this abstraction is an invariant for the loop. This invariant is then used to apply loopInvariant. Otherwise, one more iteration is initiated with loopUnwind.

7 Predicate Abstraction Scheme

The details of the predicate abstraction operator α_P have been left open in Sect. 5, because the approach does not depend on the use of any particular predicate abstraction algorithm. It is only necessary that $\varphi \rightarrow \alpha_P(\varphi)$ is always valid, and that the image of α_P is finite. Existing algorithms which can be used include those presented in [7] and in [10].

The approach has been implemented prototypically as an extension of the KeY system [1,3], a partly automated dynamic logic theorem prover for the verification of Java programs. This implementation uses the following simple predicate abstraction scheme: the abstraction of φ is the conjunction of all predicates from P which are implied by φ, i.e., $\alpha_P(\varphi) = \bigwedge\{p \in P \mid (\varphi \rightarrow p) \textit{ is found to be valid}\}$. This only allows conjunctions of the predicates, which is less flexible than supporting arbitrary boolean combinations. On the other hand, it is much cheaper to compute, which allows to handle a significantly higher number of predicates.

For efficiency, the implementation uses Simplify [8] instead of KeY itself for checking the validity of the formulas $\varphi \rightarrow p$. In order to keep the number of such checks down, known implication relationships between predicates are exploited: if $p_1 \rightarrow p_2$ is known to be valid a priori, and if we have been unsuccessful in proving $\varphi \rightarrow p_2$, then there is no need to check $\varphi \rightarrow p_1$.

Another aspect of practical importance are heuristics for automatically generating predicates. Our implementation features an ad hoc set of such heuristics. These take into consideration the program, the manually specified predicates, and the pre- and postcondition, and create in an exhaustive way many typical invariant components, such as ordering comparisons between pairs of integer variables, or that the value of a reference type variable is `null` or different from `null`. Extending such heuristics to cover more invariant elements is easily possible; however, increasing the number of predicates of course has an adverse effect on performance, so one has to strike a balance between power and efficiency.

8 Example

As an extended example, we walk through a proof for the Java implementation of *selection sort* shown in Fig. 2. The code is annotated with specifications written in the Java Modeling Language (JML) [13]. The `requires` and `ensures` clauses give a pre- and a postcondition for `sort`, respectively. The clause `diverges true` states that `sort` must not necessarily terminate; it is present because we are not concerned with termination issues in this paper.

No loop invariants are specified for the two loops of `sort`, instead only loop predicates are given. The syntax used for this has been proposed as an extension of JML in [10]: loop annotations starting with `loop_predicate` contain an arbitrary number of user-specified predicates for the loop, and free variables can be declared with `skolem_constant`. Fig. 2 gives exactly those predicates which are minimally necessary to make our implementation arrive at an invariant strong enough for proving the given method contract. These are supplemented by the

Java + JML
```
class Sorter {
    static int[] a;
    //@ public normal_behaviour
    //@ requires a != null;
    //@ ensures (\forall int x; 0 < x && x < a.length; a[x-1] <= a[x]);
    //@ diverges true;
    public static void sort() {
        //@ skolem_constant int x, y;
        //@ loop_predicate a[x] <= a[y];
        for(int i = 0; i < a.length; i++) {
            int minIndex = i;
            //@ skolem_constant int x;
            //@ loop_predicate a[minIndex] <= a[x];
            for(int j = i + 1; j < a.length; j++)
                if(a[j] < a[minIndex]) minIndex = j;
            int temp = a[i];
            a[i] = a[minIndex];
            a[minIndex] = temp;
} } }
```
Java + JML

Fig. 2. Java implementation of selection sort

heuristically generated predicates; for example, based on the specified predicate $\texttt{a[minIndex]} \leq \texttt{a[}x\texttt{]}$, the essential predicate $\forall x; (0 \leq x < \texttt{i} \rightarrow \texttt{a[minIndex]} \leq \texttt{a[}x\texttt{]})$ is generated automatically, together with many similar quantified formulas using different guards.

The JML specification can be translated into a DL sequent of the form $\varphi \vdash [\texttt{Sorter.sort();}]\psi$, where φ and ψ are essentially DL representations of the `requires` clause and the `ensures` clause, respectively. Applying the predicate abstraction proof search strategy to this root sequent yields the proof graph sketched in Fig. 3.

The first step in the construction of this proof is to perform symbolic execution of the program (abbreviated as SE in the figure) until the outer loop becomes the active statement. After applying shiftUpdate and merge (in this first iteration, to only one predecessor), we perform predicate abstraction for the outer loop. Since no fixed point has yet been reached, we unwind the outer loop, creating one branch where the loop body is entered and one where the loop terminates. The latter is immediately cut off with setBack, since it will not return to the loop entry and is therefore irrelevant for the loop invariant. On the former, the body is symbolically executed, which entails dealing with the inner loop (shown in the right half of Fig. 3) and finally leads to two branches where the outer loop is again the active statement. After applying shiftUpdate to each of them, these branches can be merged, and predicate abstraction is done again. Assuming that the resulting abstraction is not equivalent to the previous one, another identical iteration is performed.

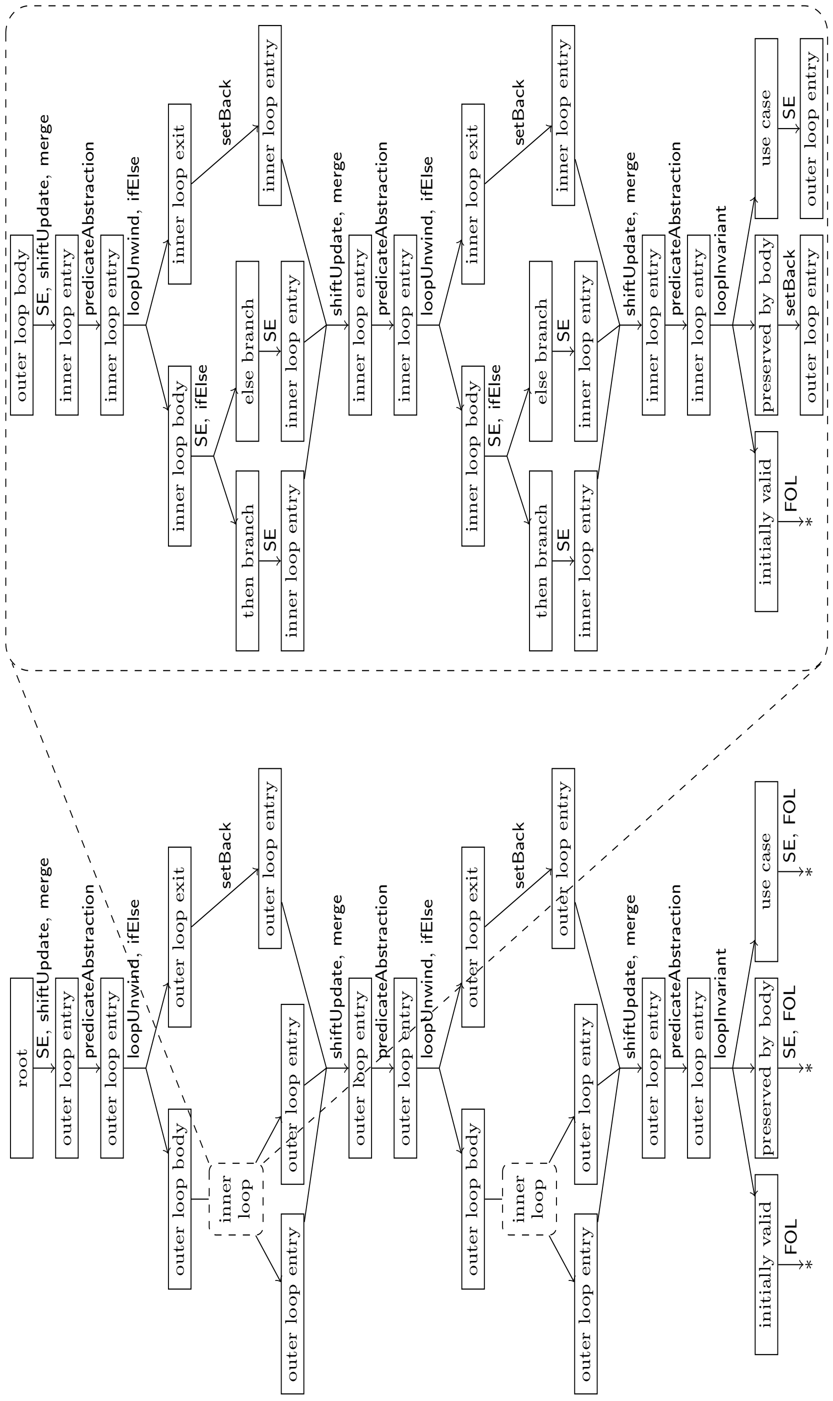


Fig. 3. Proof graph for selection sort

We assume that after this second iteration, a fixed point has been reached: the current antecedent, resulting from an application of predicateAbstraction, is logically equivalent to its counterpart in the first iteration, and is thus a loop invariant. With our implementation this inferred invariant is

$$\begin{aligned}
&\forall x; \forall y; (0 \leq x \land x < y \land y < \mathtt{i} \rightarrow \mathtt{a[}x\mathtt{]} \leq \mathtt{a[}y\mathtt{]}) \\
&\land\ \forall x; \forall y; (0 \leq x < \mathtt{i} \land \mathtt{i} \leq y < \mathtt{a.length} \rightarrow \mathtt{a[}x\mathtt{]} \leq \mathtt{a[}y\mathtt{]}) \\
&\land\ 0 \leq \mathtt{a.length}\ \land\ \mathtt{i} \leq \mathtt{a.length}\ \land\ 0 \leq \mathtt{i}\ \land\ \mathtt{a} \not\doteq \mathtt{null}\ \land\ \mathtt{exc} \doteq \mathtt{null}
\end{aligned}$$

where `exc` is a temporary variable introduced in the course of symbolic execution to buffer a possibly thrown exception. Using this for *Inv*, we apply loopInvariant. This creates three branches: the "initially valid" branch is trivial to close, because u is empty and *Inv* is identical to Γ. Proving the "preserved by body" branch entails applying loopInvariant to the inner loop, using the invariant inferred for that loop in the last iteration. As the inferred invariant is strong enough to imply the postcondition, the "use case" is closeable by further symbolic execution of the remaining program and first-order reasoning (abbreviated FOL in the figure).

The structure of the subgraph for the inner loop is analogous to the structure of the overall graph. Each time the inner loop is encountered, an invariant is inferred for it by repeated unwindings and predicate abstraction steps. The invariants inferred in the first and the second occurrence of the inner loop are different; they are dependent on the initial states occurring for the inner loop in each iteration for the outer loop. Of the three branches created by loopInvariant, the first one is again trivially closeable; the "preserved by body" branch is set back to the outer loop entry, because it does not return to that loop; and the use case is where symbolic execution actually continues back to the outer loop.

In practice, additional proof branches occur, dealing e.g. with the situation where the accessed array `a` is `null`. These are left out in Fig. 3 for simplicity. In this example, they can always be closed immediately (because the corresponding execution path is obviously infeasible), or cut off with setBack (because the execution path never returns to the respective loop entry).

9 Experiments

To give an indication of the feasibility of the approach, the results of applying the prototypical implementation to eight Java methods are listed in Table 1. For each method, the table shows its lines of combined code and specifications; the number of predicates that had to be given manually; the number of predicates that were generated automatically by the heuristics; the number of rule applications; the number of calls to Simplify for computing the predicate abstraction; and an approximate overall running time (obtained on a 1.5 GHz, 2 GB laptop).

The `getMaximumRecord` method is a simple loop which retrieves the "largest" element out of an array of objects. The second example is selection sort, as discussed in Sect. 8. The next four methods are from the Java Card API reference implementation described in [16]. These methods are simpler than selection sort

Table 1. Experimental results

	Lines	Man. prds.	Gen. prds.	Rule apps.	Simplify	Time
`LogFile::getMaximumRecord`	22	1	30	1362	41	10 s
`Sorter::sort`	22	1	1092	4594	431	90 s
`Dispatcher::dispatch`	70	0	297	2434	338	85 s
`Dispatcher::removeService`	67	1	159	3607	229	55 s
`KeyImpl::clearKey`	74	1	105	1777	252	115 s
`KeyImpl::initialize`	69	1	104	1746	242	95 s
`IntervalSeq::incSize`	33	2	178	3666	231	120 s
`Subject::registerObserver`	36	2	185	4431	242	125 s

algorithmically, but more technically involved. The last two examples are the two methods requiring loop invariants in the tutorial [1].

In all listed cases, the found invariant was strong enough to complete the verification task at hand (except for proving termination), without interaction. Manually specifying the necessary zero to two loop predicates appeared notably easier than having to provide the invariant as a whole. On the negative side, there are three additional loops in [16] for which a strong enough invariant could not be inferred. Two of them require invariants of a form (involving, e.g., existentially quantified subformulas) which are not covered by the implemented predicate abstraction scheme. The third contains deeply nested case distinctions in the loop body, which lead to large disjunctive formulas that overwhelmed Simplify.

10 Conclusions

This paper has investigated an approach for integrating abstract interpretation techniques, in particular predicate abstraction, into a calculus for deductive program verification. This allows to take advantage of the power of a deductive framework, while selectively introducing the approximation characteristic for abstract interpretation to find loop invariants automatically when necessary.

The approach consists of adding a small number of additional rules, and a dedicated proof search strategy to drive the invariant inference process. As is common for abstract interpretation, this process always finds an invariant for a loop, but this invariant is not in all cases expressive enough to be useful. Its strength heavily depends on the underlying set of loop predicates, whose elements are either generated heuristically or provided manually instead of the loop invariants themselves.

Experience with an implementation in the KeY system demonstrates the general feasibility of the approach. A line of future work is combining it with more sophisticated predicate abstraction algorithms and heuristics for generating predicates. Another possible direction is the integration of an abstraction-refinement mechanism, which would aim at systematically deriving predicates from failed proof attempts. Also, it should be possible to generalise the approach to also support other abstract domains, in addition to predicate abstraction.

References

1. Ahrendt, W., Beckert, B., Hähnle, R., Rümmer, P., Schmitt, P.H.: Verifying object-oriented programs with KeY: A tutorial. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2006. LNCS, vol. 4709, pp. 70–101. Springer, Heidelberg (2007)
2. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
3. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. The KeY Approach. LNCS, vol. 4334. Springer, Heidelberg (2007)
4. Beckert, B., Platzer, A.: Dynamic logic with non-rigid functions: A basis for object-oriented program verification. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS, vol. 4130, pp. 266–280. Springer, Heidelberg (2006)
5. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
6. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL 1977, pp. 238–252. ACM Press, New York (1977)
7. Das, S., Dill, D.L., Park, S.: Experience with predicate abstraction. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 160–171. Springer, Heidelberg (1999)
8. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A theorem prover for program checking. Journal of the ACM 52, 365–473 (2005)
9. Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. IEEE Transactions on Software Engineering 27, 99–123 (2001)
10. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: POPL 2002, pp. 191–202. ACM Press, New York (2002)
11. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
12. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12, 576–580 (1969)
13. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. ACM Soft. Eng. Notes 31, 1–38 (2006)
14. Leino, K.R.M., Logozzo, F.: Loop invariants on demand. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 119–134. Springer, Heidelberg (2005)
15. Leino, K.R.M., Logozzo, F.: Using widenings to infer loop invariants inside an SMT solver, or: A theorem prover as abstract domain. In: WING 2007 (2007)
16. Mostowski, W.: Fully verified Java Card API reference implementation. In: Beckert, B., Beckert, B. (eds.) VERIFY 2007, vol. 259, pp. 136–151. CEUR-WS.org (2007)
17. Rümmer, P.: Sequential, parallel, and quantified updates of first-order structures. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS, vol. 4246, pp. 422–436. Springer, Heidelberg (2006)
18. Schmitt, P.H., Weiß, B.: Inferring invariants by symbolic execution. In: Beckert, B. (ed.) VERIFY 2007, vol. 259, pp. 195–210. CEUR-WS.org (2007)
19. Tiwari, A., Gulwani, S.: Logical interpretation: Static program analysis using theorem proving. In: Pfenning, F. (ed.) CADE 2007. LNCS, vol. 4603, pp. 147–166. Springer, Heidelberg (2007)

Mechanised Translation of Control Law Diagrams into *Circus*

Frank Zeyda and Ana Cavalcanti

Department of Computer Science, University of York, UK
{zeyda,ana}@cs.york.ac.uk

Abstract. Previously we proposed a strategy for translating control law diagrams into *Circus*. Combining elements from Z, CSP, and a refinement calculus, *Circus* captures functional and dynamic aspects of a diagram, and allows us to formally verify implementations. The main contributions of this paper are first to discuss a generalisation of the existing translation strategy, motivated by its mechanisation and application to sizable examples. Secondly, we present a tool, the *Circus* Producer, which automates the translation, and describe how its architecture facilitates subsequent development of further verification tools.

Keywords: Simulink, verification, ClawZ, Z, CSP.

1 Introduction

Control law diagrams are commonly used by engineers in the specification and design of control systems. They describe a system as a directed graph of blocks carrying out elementary functions, and interconnecting wires transmitting data values between the outputs and inputs of the blocks. The diagrams can be structured in that (subsystem) blocks at a higher level can be defined in terms of subordinate diagrams at a lower level. The outputs of a diagram are repetitively computed in cycles of execution where inputs are taken and outputs calculated.

Where control law diagrams are used in the context of safety-critical systems, methods for analysis and validation are vital. Simulink [19] is a *de facto* standard for specifying control law diagrams and offers support for static analysis and simulation. Approaches based on model-checking have also been successfully used to verify properties of discrete-time and hybrid systems [18,10,11].

Most existing work indeed focuses on validating properties of diagrams; complementary to this, our concern is to verify the correctness of implementations. For this purpose, the ClawZ suite of tools [2,1] has been developed and successfully used in industry. ClawZ verifies implementations by constructing a functional Z model of the diagram, and deriving a refinement conjecture for a given implementation. The implementation is typically written in a subset of Ada. Discharging the refinement conjecture is achieved in ProofPower-Z, a mechanical theorem prover for the Z language; it is performed mostly automatically.

A restriction of ClawZ is that it ignores the potential parallelism between the blocks of a diagram. In principle, the computations they define can be performed in parallel, with order imposed only by the way in which they are wired. Parallelism also surfaces when independent flows of execution within subsystems give rise to the possibility of outputs being produced before all inputs are received. Even some basic blocks, such as the UnitDelay, which delays a signal by one cycle, may produce their output prior to receiving their input.

To capture this aspect of control laws, we proposed an alternative technique based on *Circus* [6], a language for refinement that incorporates elements of Z, CSP, Dijkstra's Guarded Command Language and Morgan's refinement calculus [20,7]. It is suitable for development of state-rich, reactive systems [14,15]. In our *Circus* model of Simulink diagrams, we enrich the Z model produced by ClawZ to capture parallelism. The success of ClawZ in the avionics sector to reduce the cost of verification [1] strengthens our claim that a similar approach with equal benefit can be realised using *Circus* as a formal description language.

Due to the complexity and size of diagrams in real applications, tool support is indispensable to effectively generate the models that we propose. The main contribution of this paper is to report on a tool that mechanises the translation of discrete-time[1] single-rate Simulink diagrams into *Circus* specifications.

A further contribution is a generalisation of the translation strategy as presented in [6]. We obtain (a) more flexibility in defining the structure of the *Circus* models to match that of the proposed implementations and thereby facilitate the verification, (b) cover more sophisticated wiring, that is, those in which diagram outputs may refer to the same wire; and (c) simplify the interplay between Z and *Circus* to minimise the risk of introducing errors in the Z model.

In Section 2 we present ClawZ and give an overview of *Circus* and its verification technique for control systems; we also discuss how our tool integrates with the ClawZ framework. In Section 3 we explain the extensions that we propose to the translation strategy. Section 4 then describes the use of our translation tool, the *Circus* Producer, and Section 5 addresses some design and implementation issues. Finally, in Section 6 we draw our conclusions and address future work.

2 ClawZ and *Circus*

The verification process supported by ClawZ, and its major components are depicted in Fig. 1. First, the Simulink model is submitted to the Z Producer which generates the Z model encoded in the notation of ProofPower-Z. Each block or subsystem is defined by a schema introducing variables for the inputs, outputs and state components of the block. The set of bindings of the schema specifies the behaviour of the block. The schemas for subsystems are dynamically constructed upon translation, but those for primitive blocks are inferred from a predefined library that may be extended by the user.

From the Z specification and the Ada implementation the RSG tool (Refinement Script Generator) constructs a compliance argument: a series of refinement conjectures which, if proved valid, establish the correctness of the

[1] Discrete time is a requirement for diagrams to be implementable in software.

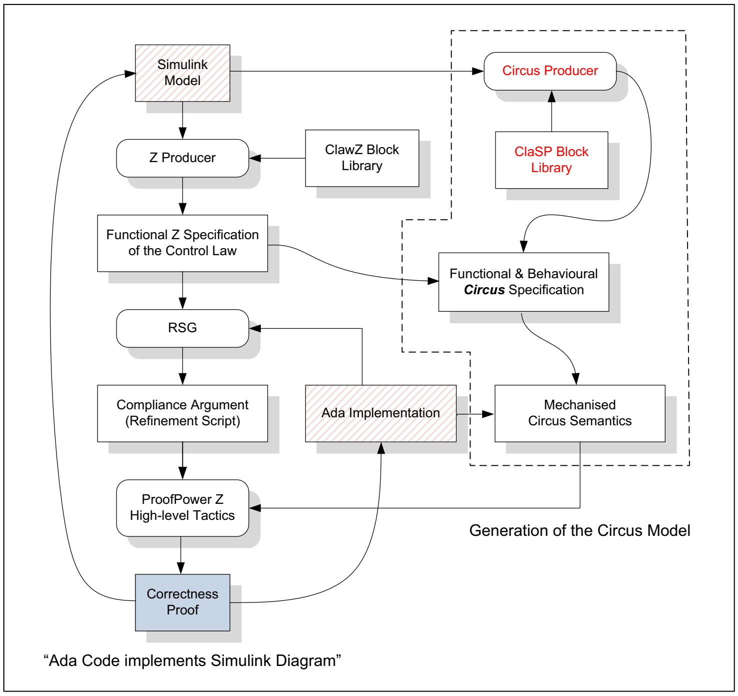

Simulink Model
Z Producer
ClawZ Block Library
Functional Z Specification of the Control Law
RSG
Compliance Argument (Refinement Script)
Ada Implementation
ProofPower Z High-level Tactics
Correctness Proof
"Ada Code implements Simulink Diagram"
Circus Producer
ClaSP Block Library
Functional & Behavioural Circus Specification
Mechanised Circus Semantics
Generation of the Circus Model

Fig. 1. ClawZ tools framework and integration of the *Circus* Producer

implementation. Using ProofPower-Z and specialised proof tactics, these conjectures are proved almost entirely automatically — even for large real systems.

In our approach and tool, we reuse elements of ClawZ to reduce the development effort, and take advantage of validated tools and Z models. We directly incorporate the schemas generated by the Z Producer into our *Circus* model. In Fig. 1, the dotted line indicates the components that we added to the ClawZ toolset to cater for *Circus* models. An additional element of supplied information is the ClaSP block library. It contains essential information regarding the concurrent behaviour of primitive blocks. Like with the ClawZ library, we anticipate that for particular diagrams the ClaSP library may have to be extended. The mechanised *Circus* semantics [16] enables us to translate the *Circus* model into a ProofPower-Z encoding, and like in the ClawZ verification process we will use specialised high-level tactics to automate the refinement proof.

Circus adopts elements from sequential programming as well as process algebra. The fundamental constructs are channels, processes and actions [20,7]. Channels are introduced through channel declarations and are required for communication and synchronisation as in CSP. Processes can be defined either explicitly or in terms of process operators. An explicitly defined process is a sequence of paragraphs that specify its state, auxiliary actions, which use or change the state information, and a main action that defines the behaviour of the process.

$$
\begin{array}{l}
\mathbf{process}\ Debounce\_Process \mathrel{\widehat=} (\\
\quad Debounce\_\_LogicalOperator\_Process \\
\qquad \{\!|\ Flag, Debounce\_\_LogicalOperator\_out, end\_cycle\ |\!\} \\
\qquad\qquad \| \\
\quad Debounce\_\_DataTypeConversion1\_Process \\
\qquad \{\!|\ Debounce\_\_LogicalOperator\_out, \\
\qquad Debounce\_\_DataTypeConversion1\_out, end\_cycle\ |\!\} \\
\qquad\qquad \| \\
\qquad \ldots \\
\quad Debounce\_\_Terminator\_Process \\
\qquad \{\!|\ Debounce\_\_SzRFlipzFlop\_out2, end\_cycle\ |\!\}) \setminus \\
\qquad\quad (Debounce\_\_LogicalOperator\_out, \\
\qquad\quad Debounce\_\_DataTypeConversion1\_out, \\
\qquad\quad Debounce\_\_DataTypeConversion2\_out,\ \ldots\ )
\end{array}
$$

Fig. 2. *Circus* translation of the Debounce diagram presented in Fig. 3

Fig. 5 provides an example of an explicitly defined process and some prior channel declarations. In the process, we first introduce a state paragraph that declares two components, namely, $Debounce\_\_Counter64Hz1\_\_UnitDelay1\_state$ and $Debounce\_\_Counter64Hz1\_\_UnitDelay2\_state$; the set $\mathbb{U}$ represents a universal type in ProofPower-Z. Actions are defined by schemas, like in the case of $Calculate\_Debounce\_\_Counter64Hz1$, or by a mixture of sequential and CSP constructs, like in the case of $Execute\_Time$. CSP operators such as guarding, parallelism, interleaving and hiding can also be used. In $Execute\_Time$, for example, we moreover use variable declarations and assignments.

Circus also provides operators to combine processes. In the models of control law diagrams, we only require parallelism, channel renaming and hiding. Fig. 2 exemplifies the n-way alphabetised parallel operator where each process is associated with a set of channels on which it is required to synchronise. Renaming changes the names of channels within a process. Finally, hiding, also used in Fig. 2, internalises communication events over given channels.

To perform the translation from Simulink to *Circus* we further require a graph model of the diagram. It records the number of inputs and outputs, and independent flows of execution of each block. It additionally includes details of whether flows depend on enabling signals, or the order of arrival of their inputs.

The *Circus* model of a diagram defines channels to represent each of its inputs, outputs and internal wires, and a basic explicitly defined process for each block. In addition, the diagram itself is modelled by a parallel composition that combines the processes defining the blocks. For example, Fig. 2 sketches the translation of the Debounce diagram given in Fig. 3.

Each of the parallel processes results from the translation of one block in the diagram. The synchronisation sets include their interface, that is, input and output signals, as well as end_cycle. While the synchronisation on interface channels corresponds to the passing of signals between blocks, synchronisation on *end_cycle* ensures that a new cycle is commenced only after all blocks have finished their computation for the current cycle. The channels that correspond to internal wires are hidden to reflect the view of the diagram as a 'black box'.

The blocks are each translated to a centralised, explicitly defined process that lifts the functional Z specification produced by ClawZ; Fig. 5 gives an example: the translation of Counter64Hz1. We maintain the functional behaviour defined by ClawZ, but also accommodate the intrinsic parallelism. The rôle of the *Flow* action is to specify, through interleaving, the independent signal flows inside the block; in our example, though, there is only one flow. For blocks with state, the *Circus* process introduces a state paragraph. The purpose of the *StateUpdate* action is then to update the state; it is based on the ClawZ schema. In Fig. 5, for conciseness, we omit the ClawZ schema $Debounce\_Counter64Hz1$.

In the next section we explain the modifications and extensions to this strategy that have been suggested by its mechanisation.

3 Extended Translation Strategy

The experience we gained from implementing and applying our tool has given rise to three extensions to the translation strategy, which we discuss in the sequel.

Structure of Models. The translation strategy outlined above distinguishes the translation of the top-level diagram from that of its blocks. The diagram is represented by a parallel composition of block models, which are centralised, explicitly defined processes. This is appropriate if parallelism in the implementation is only at the level of procedures implementing block functionality.

If, on the other hand, subsystem blocks in the top-level diagram, or any other level of the diagram structure, are implemented by parallel procedures, representing them as parallel processes is more appropriate. This renders the architecture of the specification closely aligned to that of the implementation, and greatly reduces the effort in deriving and discharging the refinement conjecture.

Centralised *Circus* processes which are implemented by parallel procedures have to be decomposed during refinement. The strategy that can accomplish this is not as easily automated, as it requires the definition of coupling invariants relating the state of the centralised and the decomposed processes.

We propose that each subsystem block of the diagram may be *selectively* translated into either an explicitly defined, centralised process, or a parallel process. The decision can be governed by the architecture of a prospective implementation. The choices have an impact only on the automation of the verification; the models that are produced as the result of the different choices are semantically equivalent, and as a consequence they both capture the intrinsic parallelism in the diagram. This can be easily proved using laws of *Circus*.

To illustrate this point, we consider the Debounce control law given in Fig. 3. (This control system filters out a potential succession of quick oscillations upon toggling the state of a mechanical sensor or switch.) In the existing translation strategy, each element of the Debounce diagram would be translated into a

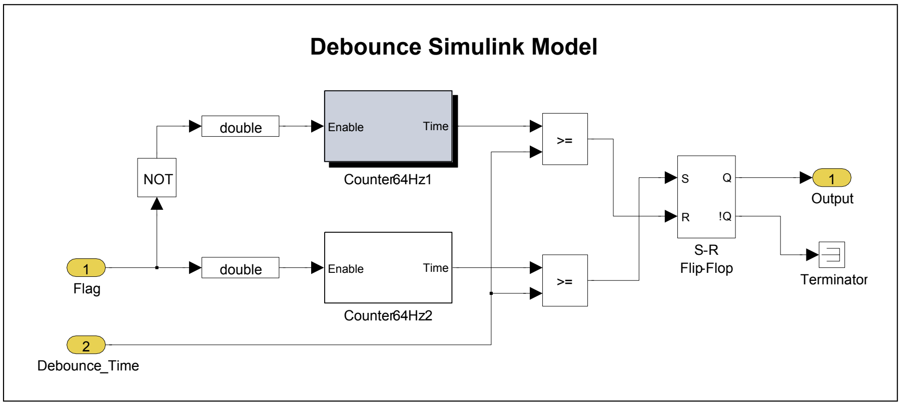


Fig. 3. Simulink diagram of the Debounce control law

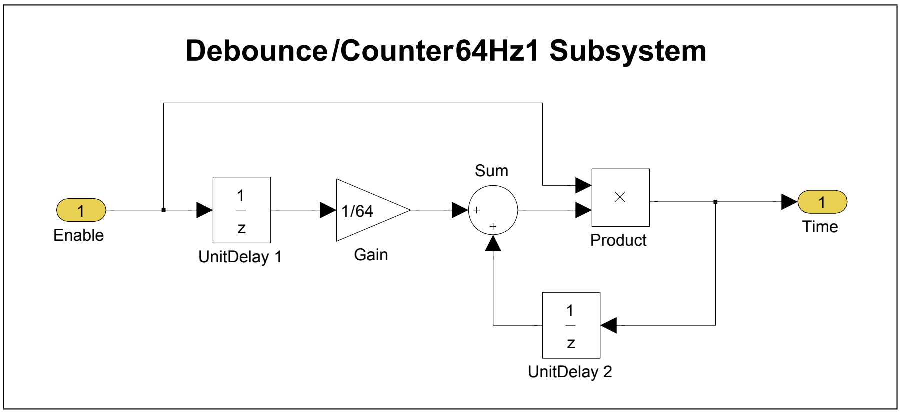


Fig. 4. Simulink diagram of the Counter64Hz1 subsystem

centralised *Circus* process. Taking, for example, the Counter64Hz1 subsystem block, included in Fig. 4, the corresponding translation would read as shown in Fig. 5.

A possible implementation may choose to first compute the Time output signal, and then proceed by *concurrently* updating the state of the two UnitDelay blocks. Clearly, this parallelism is not directly reflected in the *StateUpdate* action of the centralised *Circus* model in Fig. 5. Using our extended translation strategy, we may decide to translate this subsystem in a parallel manner. It adopts the same mode of translation we already used for the top-level diagram in the existing strategy (Fig. 2), but applies it to a subsystem. The process parallelism between the block translations exactly reflects our intention, for example, of implementing the UnitDelay1 and UnitDelay2 blocks by individual procedures.

$\textbf{channel } \mathit{Enable}, \mathit{Time} : \mathbb{U}$

$\textbf{process } \mathit{Debounce\_\_Counter64Hz1\_Process} \mathrel{\widehat=} \textbf{begin}$

$$\begin{array}{l}
\textbf{state } \mathit{Debounce\_\_Counter64Hz1\_State} == \\
\quad [\mathit{Debounce\_\_Counter64Hz1\_\_UnitDelay1\_state} : \mathbb{U}] \land \\
\quad [\mathit{Debounce\_\_Counter64Hz1\_\_UnitDelay2\_state} : \mathbb{U}]
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Init}} \\
\mathit{Debounce\_\_Counter64Hz1\_State}' \\
\hline
(\exists\, b : \mathit{Debounce\_\_Counter64Hz1\_\_UnitDelay1} \bullet \\
\quad \mathit{Debounce\_\_Counter64Hz1\_\_UnitDelay1\_state}' = b.\mathit{initial\_state}) \land \\
(\exists\, b : \mathit{Debounce\_\_Counter64Hz1\_\_UnitDelay2} \bullet \\
\quad \mathit{Debounce\_\_Counter64Hz1\_\_UnitDelay2\_state}' = b.\mathit{initial\_state})
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Calculate\_Debounce\_\_Counter64Hz1}} \\
\mathit{In1}? : \mathbb{U} \\
\mathit{Out1}! : \mathbb{U} \\
\hline
(\exists\, b : \mathit{Debounce\_\_Counter64Hz1} \bullet \\
\quad \mathit{In1}? = b.\mathit{In1}? \land \mathit{Out1}! = b.\mathit{Out1}! \land \\
\quad \mathit{Debounce\_\_Counter64Hz1\_\_UnitDelay1\_state} = b.\mathit{UnitDelay1}.\mathit{state} \land \\
\quad \mathit{Debounce\_\_Counter64Hz1\_\_UnitDelay1\_state}' = b.\mathit{UnitDelay1}.\mathit{state}' \land \\
\quad \mathit{Debounce\_\_Counter64Hz1\_\_UnitDelay2\_state} = b.\mathit{UnitDelay2}.\mathit{state} \land \\
\quad \mathit{Debounce\_\_Counter64Hz1\_\_UnitDelay2\_state}' = b.\mathit{UnitDelay2}.\mathit{state}')
\end{array}$$

$$\begin{array}{l}
\mathit{Calculate\_Time} == \\
\quad \mathit{Calculate\_Debounce\_\_Counter64Hz1} \setminus (\\
\qquad \mathit{Debounce\_\_Counter64Hz1\_\_UnitDelay1\_state}', \\
\qquad \mathit{Debounce\_\_Counter64Hz1\_\_UnitDelay2\_state}') \land \\
\quad \Xi\, \mathit{Debounce\_\_Counter64Hz1\_State}
\end{array}$$

$$\begin{array}{l}
\mathit{Execute\_Time} \mathrel{\widehat=} \\
\quad \textbf{var } \mathit{In1} : \mathbb{U} \bullet \mathit{Enable}?x \to \mathit{In1} := x \,⨟ \\
\qquad \textbf{var } \mathit{Out1} : \mathbb{U} \bullet \mathit{Calculate\_Time} \,⨟\, \mathit{Time}!\mathit{Out1} \to \mathit{Skip}
\end{array}$$

$\mathit{Flows} \mathrel{\widehat=} \mathit{Execute\_Time}$

$$\begin{array}{l}
\mathit{Calculate\_Debounce\_\_Counter64Hz1\_State} == \\
\quad \mathit{Calculate\_Debounce\_\_Counter64Hz1} \setminus (\mathit{Out1}!)
\end{array}$$

$$\begin{array}{l}
\mathit{StateUpdate} \mathrel{\widehat=} \\
\quad \textbf{var } \mathit{In1} : \mathbb{U} \bullet \mathit{Enable}?x \to \mathit{In1} := x \,⨟ \\
\qquad \mathit{Calculate\_Debounce\_\_Counter64Hz1\_State}
\end{array}$$

$$\begin{array}{l}
\bullet\ \mathit{Init} \,⨟ \\
\quad \mu\, X \bullet \mathit{Flows} \, [\![\emptyset \mid \{\!|\, \mathit{Enable} \,|\!\} \mid \{ \\
\qquad\quad \mathit{Debounce\_\_Counter64Hz1\_\_UnitDelay1\_state}, \\
\qquad\quad \mathit{Debounce\_\_Counter64Hz1\_\_UnitDelay2\_state}\} \,]\!] \, \mathit{StateUpdate} \,⨟ \\
\qquad \mathit{end\_cycle} \to X \\
\textbf{end}
\end{array}$$

Fig. 5. Centralised translation of the Counter64Hz1 subsystem

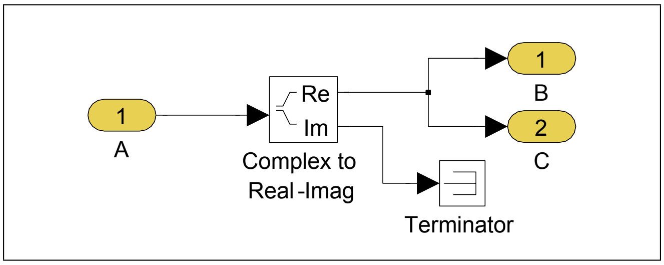


Fig. 6. Diagram causing problems for generating signal names

Naming of Signals. Internal signals of a diagram are named according to the source block they connect. Clearly, there can only be one such block for each wire, although blocks can have multiple outputs. For blocks with only one output, the corresponding signal name is obtained by appending the suffix '`_out`'; for blocks with more than one output, the suffixes '`_out1`', '`_out2`', etc. are used.

An exception to the above rule are signals that connect input and output ports of the diagram. These are always named according to the respective port they connect in the model, i.e. `A`, `B` and `C` in Fig. 6. For input signals this proves not to be an issue, since there can only be one input port acting as the source.

For output signals, Fig. 6 depicts a scenario in which the signal name for the wire connecting Complex to Real-Imag, B and C cannot be uniquely derived as there exist two output ports potentially determining the name. To solve the problem, signal names are now always determined by their source location. In the example above, the name of the signal connecting the two output ports is `SignalNamingIssue__ComplextoRealzImag_out1`. We still, however, need to introduce signals for the outputs of the subsystem, through which it communicates with other blocks when instantiated in some diagram context. Hence there will be three channel declarations for our example.

$$\textbf{channel}\ \mathit{SignalNamingIssue\_\_ComplextoRealzImag\_out}1, B, C : \mathbb{U}$$

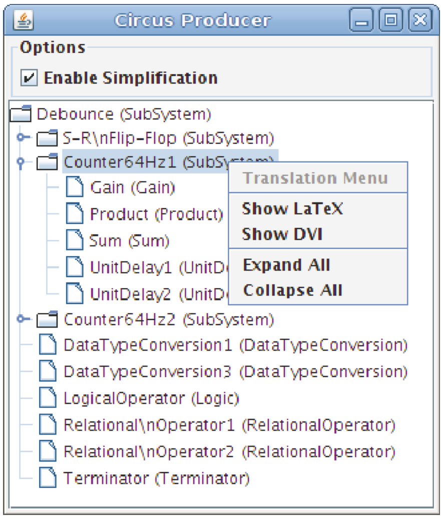
To communicate values to the output ports we take a view of them as *blocks* that simply pass on their input signal. This results in additional processes being created for each output port in the *Circus* translation, but the approach yields a very uniform treatment compatible with the fact that output ports are indeed represented as (Outport) blocks in the Simulink diagram.

Global Inclusion of the ClawZ Schemas. Finally, we avoid the inclusion of the ClawZ schemas in the local scope of the explicitly defined *Circus* processes. Initially this ensured that the Z schemas were only available in the scope in which they were used. It was an appropriate use of the modularity afforded by processes, but, for automatically generated models, it is not much of an advantage, and breaks the traceability between the *Circus* model and the ClawZ output. The actual inclusion of the ClawZ schemas takes place at a later stage when the *Circus* processes are semantically encoded into ProofPower-Z (see Fig. 1). Here, we directly incorporate the Z schemas from the respective ProofPower database generated by ClawZ upon model construction. This removes the need for reparsing the ClawZ-generated schemas when producing the *Circus* model and thereby erases the possibility of introducing errors in combining the Z and *Circus* models.

4 The *Circus* Producer

In this section we describe the main features of the *Circus* Producer — our tool for translating Simulink diagrams into *Circus*. We focus on the *usage* of the tool; its design and implementation are discussed in the next section.

User Interface. The graphical interface of the *Circus* Producer is shown in Fig. 7. The structure of the given Simulink diagram is rendered as a tree where each internal node corresponds to a subsystem, and each leaf to a primitive block. The textual description for each node gives the name of the respective element in the diagram, and in parentheses its block type. The user may double-click on internal nodes to expand or collapse them. The functions **Expand All** and **Collapse All** expand or collapse all descending nodes of the current selection.



Debounce (SubSystem)
S-R\nFlip-Flop (SubSystem)
Counter64Hz1 (SubSy
Gain (Gain)
Product (Product)
Sum (Sum)
UnitDelay1 (UnitD
UnitDelay2 (UnitD
Counter64Hz2 (SubSystem)
DataTypeConversion1 (DataTypeConversion)
DataTypeConversion3 (DataTypeConversion)
LogicalOperator (Logic)
Relational\nOperator1 (RelationalOperator)
Relational\nOperator2 (RelationalOperator)
Terminator (Terminator)

Fig. 7. Screen shot of the *Circus* Producer application

An important feature of the graphical interface is that *translation of the Simulink model is affected by the configuration of nodes in the tree control*: subsystem nodes that are expanded are translated in a parallel manner, and those collapsed are translated into centralised processes.

The context menu function **Show LaTeX** performs the translation of the (sub)diagram implied by the currently selected node; as a result, a LaTeX file is produced. The LaTeX directives used in the encoding are those of the *Circus* extension of the Community Z Tools (CZT).

By using LaTeX as the encoding language, viewable documents are easily generated for the produced processes. In fact, the **Show DVI** context menu function performs the translation as before, and additionally converts the generated LaTeX file into a DVI document and immediately displays it. The *Circus* process given in Fig. 5 was indeed automatically generated applying the *Circus* Producer to the subdiagram Counter64Hz1 of the Debounce model (Fig. 3, 4).

The ClaSP Library. Part of the translation algorithm constructs a graph model of the diagram. Signal flows for subsystem blocks are calculated dynamically. For primitive blocks, however, we need to specify the number of input and output ports, and signal flows. This information is included in the ClaSP block library.

To encode the library we use an XML-based format. The advantages of XML are, beyond standardisation and interchangeability, that the encoded file can be validated against the XML schema that defines its exact structure.

In general, the user only needs to make additions to the ClaSP library when blocks are encountered which the *Circus* Producer cannot translate. To help with their identification for a particular model, the *Circus* Producer generates warning messages each time a block is found that is not present in the ClaSP library.

To give an example of the format of library entries, Fig. 8 includes the ClaSP block specification of the UnitDelayWithPreviewResettable block. This block acts as a resettable unit delay with two outputs: one for the value of the input signal in the previous cycle, and an additional one for its current value. As the delayed output does not depend on the current input, the block has two flows.

The `<BlockType>`...`</BlockType>` pair of tags introduces the new type of block. The compulsory `name` attribute has to match the respective `BlockType` field in the Simulink encoding of the block. The optional boolean `state` attribute determines whether the block has state or not.

The aggregated `<BlockWiring>`...`</BlockWiring>` tags contain information about the inputs, outputs and flows of the block. The `<inps>`...`</inps>` and `<outs>`...`</outs>` tags specify the number of input and output ports. They can have a value `varlength`, if the block has a *variable* number of inputs, or outputs. In this case, the actual number of ports is inferred upon instantiation of the block in the model. The `<flows>`...`</flows>` tags include all independent signal flows of the block as an instance of `<flow>`...`</flow>`. In the example, there are two flows: one depends on both inputs, and the other on none.

The elements for each `<flow>`...`</flow>` instance correspond to our characterisation of the ClaSP model as formalised in [6]. Thus `<enabled>` specifies whether execution of the flow depends on enabling signals, `<order>`...`</order>` states whether the order of arrival of inputs is significant, `<rinps>`...`</rinps>` determine the set of input ports, and `<pouts>`...`</pouts>` the output ports of the flow. They can be given as individual ports (`<port>1</port>`), port lists (`<portList>1 2 3</portList>`), or ranges (`<portRange from="1" to="3">`).

Circus Model Simplification. Simplification is an optional feature of the translation which may be enabled or disabled by checking or unchecking an **Enable Simplification** check-box. This function only has an effect on the translation of

```
<ClaSP>
  <BlockLibrary>
    ...
    <!-- Unit Delay with Preview Resettable (Additional Math & Discrete) -->
    <BlockType name="UnitDelayWithPreviewResettable" state="true">
      <BlockWiring>
        <inps>2</inps>
        <outs>2</outs>
        <flows>
          <flow>
            <enabled always="true"/>
            <ordered>false</ordered>
            <rinps>
              <portList>1 2</portList>
            </rinps>
            <pouts>
              <port>1</port>
            </pouts>
          </flow>
          <flow>
            <enabled always="true"/>
            <ordered>false</ordered>
            <rinps/>
            <pouts>
              <port>2</port>
            </pouts>
          </flow>
        </flows>
      </BlockWiring>
    </BlockType>
    ...
  </BlockLibrary>
</ClaSP>
```

Fig. 8. ClaSP Library extract for the UnitDelayWithPreviewResettable block

primitive blocks which do not possess state. In such cases, the strict application of the translation procedure results in introducing vacuous state paragraphs, actions and schemas for performing the state update. Enabling simplification avoids the generation of these redundant parts of the process. The simplifications performed do not have an impact on the semantics of the translation; they merely aid readability and ease subsequent mechanical formal analysis.

5 Design and Implementation

We now discuss a few of the underlying design decisions and implementation issues encountered during the development of our tool.

Integration with CZT. An important decision in the design of the *Circus* Producer was to integrate with CZT, the Community Z Tools, for the purpose of encoding and internally representing *Circus* specifications. CZT has been initially developed to provide a component library to facilitate development of Z tools [12]. Its open architecture, however, led to various extensions, including support for *Circus* [13,9]. The integration with CZT, most importantly, avoids the need for a new design and implementation of a data model for *Circus* processes, since we can readily employ CZT's Annotated Syntax Trees (ASTs).

To process ASTs of *Circus* specifications, CZT's implementation of the Visitor design pattern endows us with a powerful and flexible mechanism to traverse and transform syntax trees. This will be especially useful for generating a semantic representation of *Circus* specifications in ProofPower, as required in further stages of our work. An additional benefit of integrating with the CZT component library is that we have the option to take advantage of existing or future CZT tools, such as the *Circus* type checker and model checker. This opens up opportunities for follow-up research using alternative approaches to reason about control laws in *Circus*, while exploiting the development effort we have already invested.

Architectural Considerations. For the development of the *Circus* Producer application, we built a transparent library of well-designed, reusable data structures and components. This makes it easier to perform modifications and extensions to existing components, and, importantly, simplifies the development of new tools.

The tool has been structured into Java packages that deal separately with various aspects of data encapsulation, parsing, analysis and processing of data objects. The central data structures reside in the three sub-packages `Simulink`, `Diagram` and `ClaSP` of the enclosing package `Data` of the root application package. The classes in these packages are used to represent the Simulink file, the underlying Simulink diagram, and the ClaSP model, respectively.

The design objective was to make handling of the data objects as easy as possible for typical tasks, so that high-level functionality can be implemented with little effort. This is achieved by a tight linkage between data objects; for example, signals in flows are aware of the wire in the diagram they refer to, wires are aware of the blocks they connect, and so on. This allows information, such as signal names or the ClaSP wiring of blocks, to be computed dynamically, and reduces the data that has to be effectively managed by the application. To counteract a loss in efficiency resulting from dynamic computation, we integrated an annotation API similar to the one of CZT to cache information once computed.

Other packages provide tool components for parsing Simulink files, ClawZ library-meta files, generate Simulink diagram object representations from parsed files, and perform high-level operations on Simulink diagrams and, as we anticipate in subsequent versions, on *Circus* models. The modular architecture of these components allows for the quick prototyping and development of new tools; this, in particular, lead to the development of a collection of Java-based supplementary tools for ClawZ extending it and facilitating its use.

Representation of Blocks. Each block in a diagram is represented by an object of a particular class. This means that developers of additional tools can introduce extra methods and fields specific to certain types of Simulink blocks, if needed. For example, the `Inport` and `Outport` classes representing the input and output ports of a diagram provide methods to obtain the port number of the blocks.

A potential complication is that upon extending the ClaSP library, classes must be created for each new block. This is a task that in practice should not be negotiated to the user. Therefore, such classes are generated automatically by a utility during compilation. The instantiation of block classes through a factory makes it possible for the developer to derive from the automatically generated classes in order to attach custom functionality if required.

Generation of the Circus Translation. The various elements of the *Circus* processes resulting from the translation are described using the string template engine developed by Parr [17]. Templates are text files which contain place-holders to be 'filled in' when the template is instantiated. This isolates the static pattern of the translation from dynamic data that has to be provided to generate the concrete results such as names of processes, actions, signals, and so on.

The meta-language of the template engine defines constructs to address common cases such as generating lists within templates, conditional inclusion of text fragments, and the instantiation of one template from inside another.

We defined 22 templates to specify the translation rules. The use of string templates has so far proved very beneficial in terms of compacting the program code and facilitating changes and adjustments to the details of the translation, in particular since recompilation is not required when altering them.

6 Conclusions

In this paper we have extended the work in [6], where we describe a strategy to translate Simulink diagrams into *Circus* models, and presented tool support for mechanical translation. The extensions allow us to align the structure of the *Circus* specification with that of a given implementation, with the objective of simplifying the proof of refinement. The extended strategy also covers a wider range of wiring configurations, since it allows block outputs to be shared as diagram outputs. Finally, it simplifies the structure of the *Circus* model, and avoids the potential to introduce errors in the Z model which it aggregates.

Case Studies. The *Circus* Producer has been employed on a number of case studies of reasonable size which have served the purpose of validating the tool and evaluating its use. The examples have been provided to us by collaborators in the avionics industry, namely, EMBRAER and QinetiQ. The diagrams exhibit subsystem structure of up to 4 nesting levels, and the most complex of them contains a total of 155 elementary blocks and 14 subsystems making the construction of the *Circus* model by hand practically infeasible. Initially, applying the tool to these diagrams yielded only a partial translations due to incompleteness of the ClaSP library. At this stage, warning messages produced by the tool helped to identify the missing blocks, and subsequent consultation of the Simulink documentation to determine their behaviour in terms of flows.

The produced *Circus* models have been validated by inspection and comparison with the translation strategy in [6], and besides were tested for syntactic errors using the *Circus* parser of CZT. A further degree of validation will take place when future work semantically encodes these models in ProofPower-Z and applies the refinement strategy. We have compared the translation of the same diagram in different configurations, in particular to verify that parallelism is correctly represented in the sequential translation of subsystems. Notably, QinetiQ provided us with an example in which parallelism that surfaced at the top-most level of a digram revealed certain assumptions made about the environment by an implementation that were not explicit in the model.

As with ClawZ, the process of generating the *Circus* specification for control law diagrams can be largely automated, requiring a minimal amount of user interaction: the extension of the ClaSP library and the configuration of the translation of subsystem blocks as centralised or parallel processes. For both no knowledge of the underlying semantic details of the *Circus* representation is needed. This is very important as we would like most aspects of the verification to be driven by engineers without in-depth knowledge of the underlying verification strategy, let alone its formal justification. The *Circus* Producer, including its source code, is openly available for download from http://www.cs.york.ac.uk/circus/cld/tools.html.

Related Work. In [3], an algorithm and tool is presented to translate Simulink diagrams into formal descriptions understood by the NuSVM model checker. Models are specified as finite state machines, hence the technique only applies to diagrams with finitary state spaces. The focus of this work is on automated verification of properties about the diagram; in comparison, we are concerned with proving conformity between diagrams and their implementations.

A formal semantics and tool support to reason about functional and timing aspects of Simulink diagrams is described in [8]. This work is based on the Timed Interval Calculus (TIC); tool support is provided for mechanical translation of TIC specifications generated from Simulink diagrams into corresponding specifications to be processed by the PVS theorem prover. This is again to validate properties of the diagram rather than to verify implementations. We currently do not characterise timing properties in our semantics; future work will consider the use of the timed extension of *Circus* for this purpose.

A Lustre model of Simulink diagrams is the object of the work in [4], which reports on the development of a strategy and translator utility. This can also be regarded as a formalisation of Simulink since Lustre, like *Circus*, is equipped with a formal semantics, including strong typing. The potential of this approach to produce implementations adhering to a high standard of reliability is, for example, in the use of certified code generators for specific target languages.

Future Work. At present, the automatic translation does not yet account for enabling signals which govern enabled, trigger or action subsystems. In [6] we already described how they should be handled, however work to extend ClawZ and the *Circus* Producer to accommodate these rules is still pending.

Another line of future work is the translation of StateFlow blocks, which permit the specification of subsystems using a notation based on State Charts. In [5], we described how *Circus* can be used to define models of StateFlow diagrams that can be used as components of a model of a Simulink diagram which includes them; automation is our next step.

The next phase of our work aims to translate the tool-generated *Circus* specifications into corresponding ProofPower-Z specifications using our semantic embedding of *Circus* proposed in [16]. We are then able to mechanically reason, within ProofPower, about the *Circus* specification, and apply our refinement strategy to show the validity of implementations. Automating the translation into ProofPower-Z shall only pose a problem of minor difficulty, however the mechanised proof of the refinement conjecture sets a substantial challenge.

Acknowledgements. We benefitted from discussions with Phil Clayton, Leo Freitas and Daniel Bolton. We also acknowledge EPSRC for funding this work under the Programming from Control Laws research grant EP/E025366/1.

References

1. Adams, M., Clayton, P.: ClawZ: Cost-Effective Formal Verification of Control Systems. In: Lau, K.-K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, pp. 465–479. Springer, Heidelberg (2005)
2. Arthan, R., Caseley, P., O'Halloran, C., Smith, A.: ClawZ: Control laws in Z. In: 3^{rd} International Conference on Formal Engineering Methods, September 2000, pp. 169–176. IEEE Computer Society Digital Library (2000)
3. Meenakshi, B., Bhatnagar, A., Roy, S.: Tool for Translating Simulink Models into Input Language of a Model Checker. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 606–620. Springer, Heidelberg (2006)
4. Capsi, P., Curic, A., Maignan, A., Sofronis, C., Tripakis, S.: Translating Discrete-Time Simulink to Lustre. In: Alur, R., Lee, I. (eds.) EMSOFT 2003. LNCS, vol. 2855, pp. 84–99. Springer, Heidelberg (2003)
5. Cavalcanti, A.: Stateflow Diagrams in Circus. In: SBMF 2008, pp. 1–16 (2008)
6. Cavalcanti, A., Clayton, P., O'Halloran, C.: Control Law Diagrams in Circus. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 253–268. Springer, Heidelberg (2005)
7. Cavalcanti, A., Sampaio, A., Woodcock, J.: A Refinement Strategy for Circus. Formal Aspects of Computing 15(2–3), 146–181 (2003)
8. Chen, C., Dong, J.S.: Applying Timed Interval Calculus to Simulink Diagrams. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 74–93. Springer, Heidelberg (2006)
9. Freitas, L., Woodcock, J., Cavalcanti, A.: An Architecture for Circus Tools. In: SBMF 2007: Brazilian Symposium on Formal Methods (August 2007)
10. Krogh, B.: Approximating Hybrid System Dynamics for Analysis and Control. In: Vaandrager, F.W., van Schuppen, J.H. (eds.) HSCC 1999. LNCS, vol. 1569, p. 2. Springer, Heidelberg (1999)
11. Krogh, B.: Recent Developments in Modeling and Analysis of Hybrid Dynamic Systems. In: Donatelli, S., Kleijn, J. (eds.) ICATPN 1999. LNCS, vol. 1639, p. 106. Springer, Heidelberg (1999)
12. Malik, P., Utting, M.: CZT: A Framework for Z Tools. In: Treharne, H., King, S., Henson, M.C., Schneider, S. (eds.) ZB 2005. LNCS, vol. 3455, pp. 65–84. Springer, Heidelberg (2005)
13. Miller, T., Freitas, L., Malik, P., Utting, M.: CZT Support for Z Extensions. In: Romijn, J.M.T., Smith, G.P., van de Pol, J. (eds.) IFM 2005. LNCS, vol. 3771, pp. 227–245. Springer, Heidelberg (2005)

14. Oliveira, M., Cavalcanti, A., Woodcock, J.: Refining Industrial Scale Systems in Circus. In: Communicating Process Architectures. Concurrent Systems Engineering Series, vol. 62, pp. 281–309. IOS Press, Amsterdam (2004)
15. Oliveira, M., Cavalcanti, A., Woodcock, J.: Formal Development of Industrial-Scale Systems in Circus. Innovations in Systems and Software Engineering 1(2), 125–146 (2005)
16. Oliveira, M., Cavalcanti, A., Woodcock, J.: A UTP semantics for Circus. Formal Aspects of Computing, Online First (2007)
17. Parr, T.: StringTemplate Engine, `http://www.stringtemplate.org`
18. Ranville, S., Black, P.E.: Automated Testing Requirements — Automotive Perspective. In: The Second International Workshop on Automated Program Analysis, Testing and Verification (May 2001)
19. The MathWorks, Inc. Simulink ® (1994–2008)
20. Woodcock, J., Cavalcanti, A.: The Semantics of Circus. In: Bert, D., Bowen, J.P., Henson, M.C., Robinson, K. (eds.) B 2002 and ZB 2002. LNCS, vol. 2272, pp. 184–203. Springer, Heidelberg (2002)

Realizability of Choreographies Using Process Algebra Encodings

Gwen Salaün[1] and Tevfik Bultan[2]

[1] University of Málaga, Spain
salaun@lcc.uma.es
[2] University of California, Santa Barbara, USA
bultan@cs.ucsb.edu

Abstract. Service-oriented computing has emerged as a new programming paradigm that aims at implementing software applications which can be used through a network via the exchange of messages. Interactions among a set of services involved in a new system are described from a global point of view using *choreography* specification languages such as WS-CDL or collaboration diagrams. In this paper, we present an encoding of collaboration diagrams into the LOTOS process algebra. This encoding allows to (i) check choreography specification using the LOTOS verification toolbox (CADP), (ii) check realizability of collaboration diagrams for both synchronous communication and bounded asynchronous communication, and (iii) automate service peer generation. *Realizability* indicates whether peers can be generated from a choreography such that they will behave exactly as formalized in its specification. If the collaboration diagram is unrealizable, our approach extends the peer generation process by adding some communications that make the peers respect the choreography specification.

1 Introduction

Formal methods play a key role in many open research problems that are of significant importance to the field of service-oriented applications. This is the case for problems related to *choreography*, *i.e.*, specification of interactions among a set of services from a global point of view. Several formalisms have already been proposed to specify choreographies: WS-CDL, collaboration diagrams, process calculi, BPMN, SRML, etc. Given a choreography specification, it would be desirable if the local implementations, namely *peers*, can be automatically generated via projection. However, generation of peers that precisely implement the choreography specification is not always possible: This problem is known as *realizability.*

Recent works on this topic [10,15,4,2] advocate techniques to check the realizability of a choreography, or define well-formedness rules to be applied while writing the choreography specification in order to ensure its realizability. To the best of our knowledge, no solution has been proposed yet to generate peers realizing any choreography without adding rules or constraints on the choreography language or on specifications written with it. Accordingly, our contribution is twofold. First, our solution generates peers for any choreography specification by extending them with additional messages if the choreography is unrealizable. Second, our approach is supported by tools for verification of choreographies, testing realizability, and generation of peers in a completely automated way.

In this paper, we use collaboration diagrams as the choreography specification language. We propose an encoding of collaboration diagrams into the LOTOS process algebra. We chose LOTOS because it provides a good level of expressiveness to describe all the collaboration diagram interaction constraints, and is equipped with a rich toolbox (CADP) which offers state-of-the-art tools for state space exploration and verification. The LOTOS encoding allows to (i) verify choreography specification using CADP, (ii) check realizability of collaboration diagrams for both synchronous communication and bounded asynchronous communication[1], and (iii) automate service peer generation, adding new messages if the diagram is unrealizable.

The rest of this paper is organized as follows. Section 2 introduces collaboration diagrams and the problem of their realizability. Section 3 presents our encoding into LOTOS, and how this encoding is used to test realizability and generate peers. Section 4 sketches the tools that support our approach. Section 5 compares our proposal to related work, and Section 6 ends the paper with some concluding remarks.

2 Collaboration Diagrams

A collaboration diagram [2] (called communication diagram in UML 2) consists of a set of peers, a set of links between peers, and a set of message send events associated with links. An event is a tuple containing a dependency relation (facultative), a (unique) label, a message, and a recurrence type. Labels (1, 2, 3, ..., A1, A2, A3, ..., B1, B2, B3, ...) contain prefixes (ε, A, B) that organize events into different threads. All messages in one thread share the same prefix and execute based on the numerical order defined by their labels. Messages from different threads occur concurrently, and can be interleaved in any order that respects the dependency relation. A recurrence type is either "1" (default type) meaning that the associated event happens once, "?" for a conditional event (the event may occur once or it may not occur at all), or "*" for an iterative event (the event may not occur at all or it may occur multiple times).

Figure 1 presents a collaboration diagram for a train station service that we will use as a running example throughout this paper. This diagram contains four peers, respectively Customer, TrainStation, Availability, and Booking. It involves three threads: 1) The main thread with prefix ε and events 1 and 2; 2) The

[1] Promela (with its SPIN model-checker) is a formalism we could have used as an alternative to LOTOS, since Promela supports both synchronous and asynchronous communications whereas asynchronous communication can be expressed in LOTOS by explicitly encoding queues. However, SPIN does not provide behavioural equivalence checking we needed for the approach at hand.

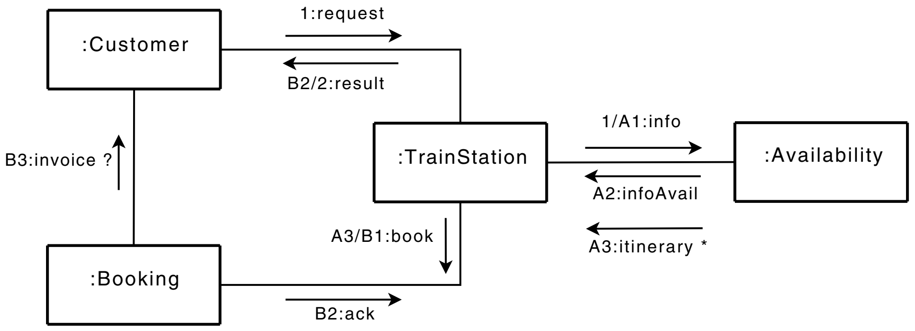


Fig. 1. Train station service collaboration diagram

A thread with prefix A and events A1, A2 and A3; and 3) The B thread with prefix B and events B1, B2 and B3. The collaboration diagram starts by the emission of a request to the train station (event 1). Next, the station checks ticket availability (events A1, A2, and A3), reserves tickets (events B1, B2, and B3), and sends the result to the customer (event 2).

Let us focus on thread A. It contains three events, the first one, 1/A1:info, means that the message info should be sent by peer TrainStation to peer Availability only after the execution of the event 1. I.e., the tuple (1, A1) is included in the dependency relation, indicating that the event A1 is executable only after the event with label 1 (namely 1:request) has been executed. The third event of thread A (A3:itinerary *) must be run after A2 (due to the sequential order within a thread), and can be executed several times (due to "*" recurrence type).

Before illustrating the realizability problem for collaboration diagrams, let us introduce the *peer model*. A peer is described as a Labeled Transition System (LTS). An LTS is a tuple (A, S, I, F, T) where: A is an alphabet, that is a set of messages with direction ("!" for emission, and "?" for reception), S is a set of states, $I \in S$ is the initial state, $F \subseteq S$ are final states, and $T \subseteq S \times A \times S$ is the transition function. Peers interact using binary communication on same message names with opposite directions. In this paper, we will consider both synchronous and asynchronous communication models.

A couple of unrealizable collaboration diagrams are presented in Figure 2. The first one (left hand side) is unrealizable because it is impossible for C to know when A sends its request message (no interaction between A and C). Hence, the peers cannot respect the execution order of messages as specified in the collaboration diagram. The second one is slightly more subtle because this diagram is realizable for synchronous communication, and unrealizable for asynchronous communication. Indeed, in case of synchronous communication, C can synchronize with A (rendez-vous) only once request is sent, so the message order is respected. This is not the case for asynchronous communication, since C sends its message to A without knowing if A has sent request or not. Therefore, the correct order between the two messages cannot be satisfied. We also give in Figure 2 the LTS generated for peer A.

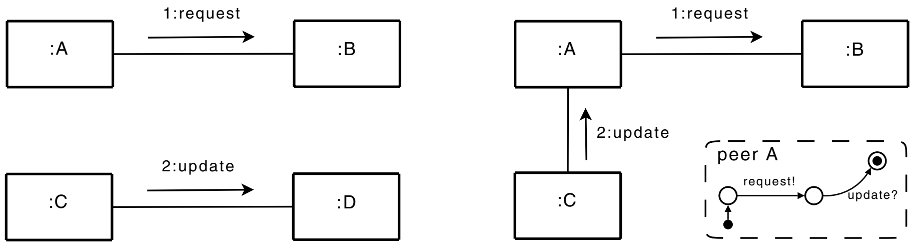


Fig. 2. Examples of unrealizable collaboration diagrams

Although realizability is easily figured out for these simple examples, it is much more complicated to say if the collaboration diagram presented in Figure 1 is realizable or not. We present in Section 3 an approach to automate the realizability test, and show that the train service collaboration diagram is unrealizable for asynchronous communication.

3 Encoding into LOTOS

The backbone of our proposal is an encoding of collaboration diagrams into the LOTOS process algebra [9]. We chose LOTOS because it relies on a rich notation that allows to specify complex concurrent systems possibly involving data types. In a second step, the LOTOS encoding allows (i) choreography verification by using model checking tools available in CADP [8], (ii) realizability test and (iii) generation of service peer implementations both for synchronous and bounded asynchronous communication models. The different steps of our approach are completely automated by different tools we present in Section 4.

3.1 Encoding Collaboration Diagrams into LOTOS

The collaboration diagram choreography is encoded using a LOTOS process. This process is split up in as many parts (referred as thread behaviour in the following) as there are threads in the diagram. Each thread behaviour encodes all the messages involved in its thread in the order in which they must be executed (achieved using the LOTOS action prefix operator). Each message is encoded using source and target peer names as prefixes to avoid name clashes. The conditional recurrence type "?" is encoded as a choice between the actual message (condition is true), and a termination (exit) meaning that the condition is false and the message not transmitted. The iterative recurrence type "*" is translated into LOTOS using an intermediate looping process whose behaviour is: message; loop_process [message] [] i ; exit.

Each thread behaviour evolves independently, and they synchronize together to respect dependency constraints (explicitly specified at the beginning of some events, *e.g.*, 1/A1:info) using new messages prefixed by "SYNC_". These messages are inserted in the LOTOS specification in two cases: (i) before running a message if this event depends on another message execution, (ii) after a message when this message appears in a dependency relation in the diagram. In the second case, the synchronization message should not block the thread execution, accordingly it is interleaved with the rest of the thread behaviour.

Let us give a part of the LOTOS code generated for our running example. We can distinguish the three threads, respectively for events starting by A, B, and numbers. Thread A for instance contains three messages (info, infoAvail, and itinerary) which are encoded in sequence and prefixed with peers participating in these interactions (only initials for readability reasons). The last message (a_ts_itinerary) involves an iterative recurrence type and is therefore translated using a loop_process. An example of "?" recurrence type is given at the end of thread B where the choice ([]) is used to express the execution of message invoice (b_c_invoice) or not (exit).

As regards synchronization between thread behaviours, we can see for instance that thread A synchronizes with the two others on messages SYNC_1 and SYNC_A3. SYNC_1 is used to synchronize thread A and the main thread in order to run message ts_a_info with label A1 after message c_ts_request with label 1 (execution of SYNC_1 acts as a pre-condition to the execution of ts_a_info). In thread B, the event B1 can only occur after A3 therefore message a_ts_itinerary (which is labeled by A3) is followed by a SYNC_A3 message, in order to run ts_b_book after A3. Note that the synchronization message SYNC_A3 should not block the thread execution. Accordingly it is interleaved with the rest of the thread behaviour (exit in this case, since it is the end of the thread).

```
(             (* -- thread A encoding -- *)
   SYNC_1;
      ts_a_info;
         a_ts_infoAvail;
            loop_process [a_ts_itinerary] >>
               ( SYNC_A3; exit ||| exit )
)
|[SYNC_1, SYNC_A3]|
(             (* -- thread B encoding -- *)
   (
      SYNC_A3;
         ts_b_book;
            b_ts_ack;
               (
                  SYNC_B2; exit
                  |||
                  (
                     b_c_invoice; exit
                     []
                     exit
                  ) >> exit
               )
```

```
        )
        |[SYNC_B2]|
        ( ...  (* -- main thread's encoding -- *) )
    )
```

From this encoding, the corresponding LTS can be generated using CADP state space generation tools, and verified using the Evaluator model-checker [13]. We checked for instance the liveness property stating that each c_ts_request is eventually answered (ts_c_result). We show in Figure 3, the LTS obtained for the collaboration diagram from the LOTOS encoding. This LTS was obtained by hiding "SYNC_" messages, and by minimizing the resulting LTS (determinization, removal of τ transitions[2], and suppression of similar paths) using reduction techniques[3] available in the CADP toolbox.

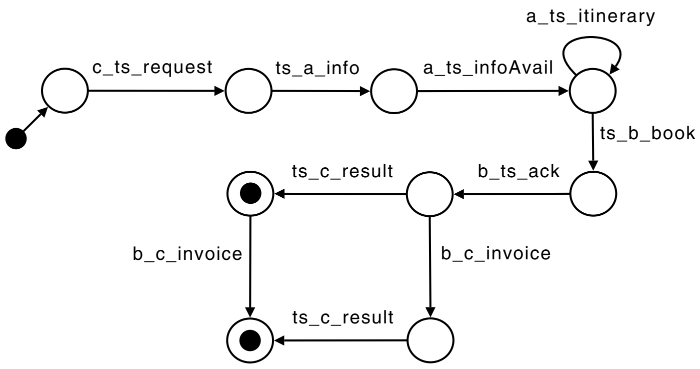


Fig. 3. Train station service: collaboration diagram LTS

3.2 Peer Generation

Peers are generated by projection from the LOTOS process encoding the collaboration diagram. This is achieved by generating a LOTOS process for each peer whose body is an instance of the collaboration diagram process, and hiding in this process all the messages in which the peer does not participate in.

Figure 4 gives a graphical view of peers generated for our running example from their LOTOS descriptions. For instance, peer Booking (Fig. 4, (b)) starts receiving a book request (ts_b_book?) from the train station, sends back an acknowledgement (b_ts_ack!), and either stops or sends an invoice to the customer (b_c_invoice!). We recall that peers interact on same message names with opposite directions, *e.g.,* c_ts_request! in the customer with c_ts_request? in the train station.

[2] τ transitions stand for internal actions. These transitions are generated while compiling the LOTOS code. For example the LOTOS sequential composition ">>" inserts such a τ transition in the corresponding state space.

[3] In this paper, minimizations are achieved using weak trace, safety and strong reductions.

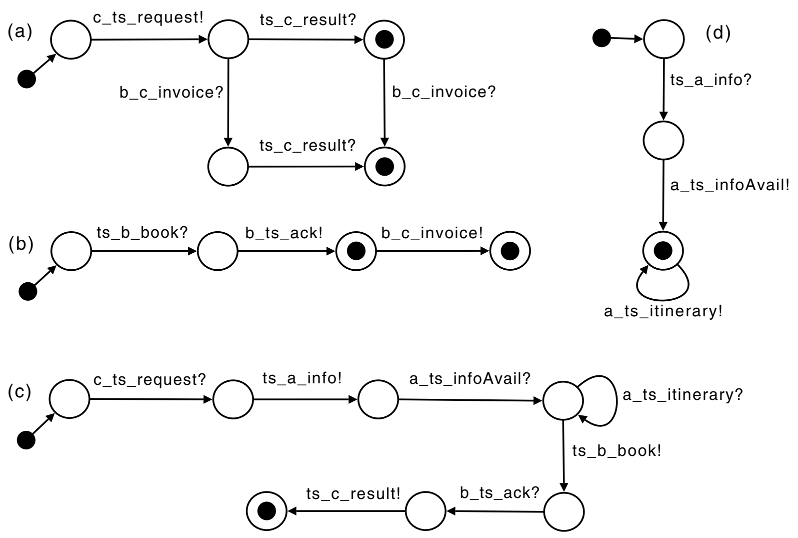


Fig. 4. Peers generated from the collaboration diagram: (a) customer, (b) booking, (c) train station, (d) availability

Once peers are generated, it is very difficult to say if their execution will respect the interaction constraints specified in the collaboration diagram (order of messages within a thread, and inter-thread message dependencies). In the next subsection, we propose automated techniques to check realizability.

3.3 Realizability

Our idea is to compute realizability by comparing the collaboration diagram LTS with the system composed of interacting peers using behavioural equivalences [14]. If these two systems are equivalent, it means that the peer generation exactly preserves the collaboration diagram constraints. If they are not, it is because peers do not generate the same interactions than those specified in the diagram, therefore it is unrealizable. Computing realizability is achieved in two steps: (i) generation of the system composed of interacting peers, and (ii) equivalence checking between the LTS resulting from step (i) and the collaboration diagram LTS. In the following, we consider both synchronous and bounded asynchronous communication models.

Synchronous communication. LOTOS relies on synchronous communication, therefore from the LOTOS code obtained previously, we generate an LTS for each peer process, and compose all peers in parallel making explicit messages on which they synchronize.

Let us now give the resulting system for our running example. This system is given in SVL [7] below. SVL is a scripting language that complements the LOTOS encoding, and automates parts of the approach by calling the different

CADP tools we reuse. Moreover, these scripts were used to circumvent the state explosion problem (see a discussion on this issue in Section 4). Bcg files (delimited by double quotes and with extension bcg below) are internal state/transition representations computed (by CADP) from the LOTOS peer processes. Message directions "!" and "?" as added in Figure 4 for pretty-printing reasons, have a different semantics in LOTOS, they are used for value passing. Since, we do not need this feature here, we have encoded messages without any direction for the synchronous case as they appear in the synchronization sets (noted between `|[..]|`) below. If two peers do not have to synchronize, they are composed using the interleaving operator (`|||`).

```
"distributed_system.bcg" =
   "peer_Customer_lts.bcg"
   |[c_ts_request, ts_c_result, b_c_invoice]|
   (
     "peer_TrainStation_lts.bcg"
     |[ts_a_info, a_ts_infoAvail, a_ts_itinerary, ts_b_book, b_ts_ack]|
     (
       "peer_Availability_lts.bcg" ||| "peer_Booking_lts.bcg"
     )
   )
```

Once this system is generated and reduced, we compare it to the collaboration diagram LTS generated as explained in Section 3.1 using a strong equivalence relation [14]. This check either says that both systems are equivalent and the collaboration diagram is then realizable, or returns *false* which means that the diagram is unrealizable. As far as our running example is concerned, the equivalence test returns *true* for synchronous communication.

Asynchronous communication. This case is slightly more difficult because asynchronous communication is not supported by LOTOS. To simulate how the system evolves with an asynchronous communication model, we generate some LOTOS code to implement bounded FIFO queues. Each peer is associated with a queue (a LOTOS process) from which it can consume messages received beforehand. This also means that a peer which wants to send a message to another one, will actually interact (synchronously) with the other one's queue. A queue process needs a queue datatype (BQueue below) to store received messages. This datatype is implemented using algebraic specification facilities provided by LOTOS. A queue process can either interact with other peers on messages that can be received by its own peer (t_s_book for the Booking queue below), or synchronizes with its own peer if this one wants to evolve by consuming a message available in its own queue (t_s_book_REC for the Booking peer). Note that a local communication between a peer and its queue is suffixed with "_REC", whereas a communication between a peer (sender) and a queue is not suffixed. The datatype encoding queues defines several operations: bisfull tests if the queue is full, binsert appends a message to the end of the queue, bishead tests if a message appears at the head of the queue, and bremove suppresses the message at the head of the queue.

```
process queue_Booking [ts_b_book, ts_b_book_REC] (q:BQueue) : exit :=
   [not(bisfull(q))] ->
      ts_b_book;
        queue_Booking [ts_b_book, ts_b_book_REC] (binsert(ts_b_book,q))
   []
   [bishead(ts_b_book,q)] ->
      ts_b_book_REC;
        queue_Booking [ts_b_book, ts_b_book_REC] (bremove(q))
   []
   exit
endproc
```

Next, a process for each couple *(peer, queue)* is generated in LOTOS. A peer and a queue interact together on all messages (suffixed with "_REC") that can be received by the peer. From an external point of view, these messages are not of interest, and that is why they are hidden. We show below the LOTOS peer_queue_Customer process body for illustration purposes. Notice that the process queue_Customer below is instantiated with a size set to 1 and no messages in the queue (nil). The queue size is an input parameter of the LOTOS encoding.

```
hide ts_c_result_REC, b_c_invoice_REC in
   (
      peer_Customer [...]
      |[ts_c_result_REC, b_c_invoice_REC]|
      queue_Customer [...] (queue (1, nil))
   )
```

Finally, the distributed system (in SVL below) is obtained by compiling all LOTOS processes encoding couples *(peer, queue)* into bcg files, and making all these couples synchronize correctly on messages exchanged among peers (that is all messages sent from peers to corresponding queues).

```
"distributed_system_async.bcg"=
   "peer_queue_Customer.bcg"
   |[c_ts_request, ts_c_result, b_c_invoice]|
   (
     "peer_queue_TrainStation.bcg"
     |[ts_a_info, a_ts_infoAvail, a_ts_itinerary, ts_b_book, b_ts_ack]|
     (
       "peer_queue_Availability.bcg" ||| "peer_queue_Booking.bcg"
     )
   )
```

Once the distributed system is computed, realizability is checked similarly to the synchronous case, by comparing if the collaboration diagram LTS obtained as presented in Section 3.1 is strongly equivalent to the distributed system.

As far as our running example is concerned, the equivalence test says *false*, and indicates that the trace c_ts_request, ts_a_info, a_ts_infoAvail, ts_b_book appears in both systems, but a_ts_itinerary is then present in the distributed system (it should not be), and not in the collaboration diagram LTS. The problem here is that the train station peer has no way to know whether the availability peer will send or not a_ts_itinerary because the recurrence type is "*" which means zero or several times. So, what happens is that the train station peer sends ts_b_book to the booking peer (assuming the availability peer will never send a_ts_itinerary), and after this emission, the availability peer finally sends a_ts_itinerary, thus the dependency relation A3/B1:book is not respected. We show in the next subsection how such unrealizable collaboration diagrams can be implemented.

3.4 Peer Generation, Extended

To make peers respect interaction constraints of unrealizable collaboration diagrams, we have to insert additional communications among peers. To do so, peers have to (i) respect the application order of messages in each thread, and (ii) respect dependency relations which specify constraints on the firing of a specific message. The first constraint is achieved by adding in the collaboration diagram encoding some explicit messages prefixed with "SEQ_" between each thread message. As regards the second one, we will use the "SYNC_" messages that have been used in the initial encoding to respect message dependency relations.

Let us illustrate with thread A of our running example, where in addition to messages SYNC_1 and SYNC_A3, two new messages SEQ_A1 and SEQ_A2 appear respectively after messages ts_a_info and a_ts_infoAvail. It is not useful to insert such a message after the last message since it is the end of the thread.

```
(               (* -- thread A encoding -- *)
   SYNC_1;
      ts_a_info;
         SEQ_A1;
            a_ts_infoAvail;
               SEQ_A2;
                  loop_process [a_ts_itinerary] >>
                     ( SYNC_A3; exit ||| exit )
) ...
```

From this extended collaboration diagram encoding, peers are generated by keeping visible the messages in which the peer does participate in, and also some of the additional communications introduced above. Additional communications to be kept are figured out following two rules: (i) A peer contains in its behaviour all "SEQ_" messages of a specific thread if the peer participates in at least one interaction of this thread; (ii) a peer contains in its behaviour each "SYNC_" message for which the corresponding message (*e.g.,* for SYNC_1, the message labeled 1 that is c_ts_request) is either one of its own messages, or is used in a dependency relation of the collaboration diagram. For both rules, peers synchronize on all additional communications that they share in their alphabets.

Let us illustrate that showing peer Booking (Fig. 5) generated with this approach. First, since peer Booking only participates in thread B, its behaviour contains messages SEQ_B1 and SEQ_B2 which means that all peers involved in

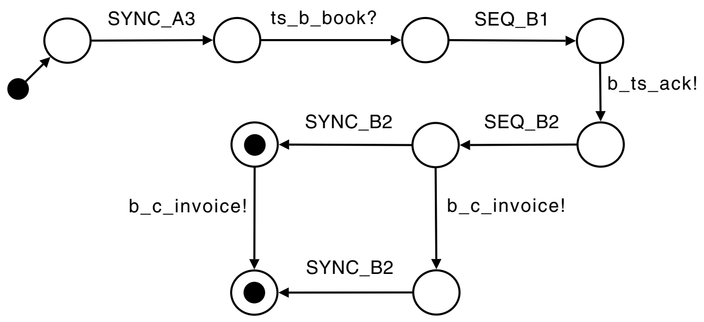


Fig. 5. Peer Booking with additional messages

thread B (namely peers Customer, Booking, and TrainStation) will synchronize using these messages so as to respect the execution order of messages in this thread. Second, two messages for dependency relations, SYNC_A3 and SYNC_B2, are used too. SYNC_A3 is necessary because message ts_b_book must be run only after the message identified by A3 (a_ts_itinerary) in the collaboration diagram. Moreover, SYNC_B2 appears in peer Booking because the message identified by B2 in the collaboration diagram (b_ts_ack) is used as dependency of another message (ts_c_result sent by the train station to the customer), thus once b_ts_ack is sent, peer Booking will interact with peers Customer and TrainStation to inform them the result can be emitted.

Once the new peers are generated, the distributed system is built by extending the description given in Section 3.3 with additional communications and also synchronizing peers on them. We recall that all peers do not synchronize on all additional communications but only on those belonging to their alphabet and shared with the other peers. Finally, equivalence between the collaboration diagram LTS and the distributed system in which all additional communications have been hidden, confirms that the extended peers conform to the collaboration diagram.

4 Tool Support and Experiments

The different steps of our approach are completely automated by several tools (Fig. 6). We have implemented a prototype tool named cd2lotos which, given a collaboration diagram, generates the LOTOS code necessary to compute all the results we have presented before in this paper. The cd2lotos prototype also generates some SVL scripts that complement the LOTOS encoding and automate the rest of the process by calling the different CADP tools we reuse. Thus, LTS generation is achieved using Caesar.adt and Caesar LOTOS compilers, as well as reduction techniques available in Reductor. Model-checking can be performed using Evaluator. Note that model-checking is the only step which is not fully automated. Indeed, if a designer wants to go beyond basic checks (such as deadlock-freeness), (s)he has to write some formulas encoding properties to be satisfied

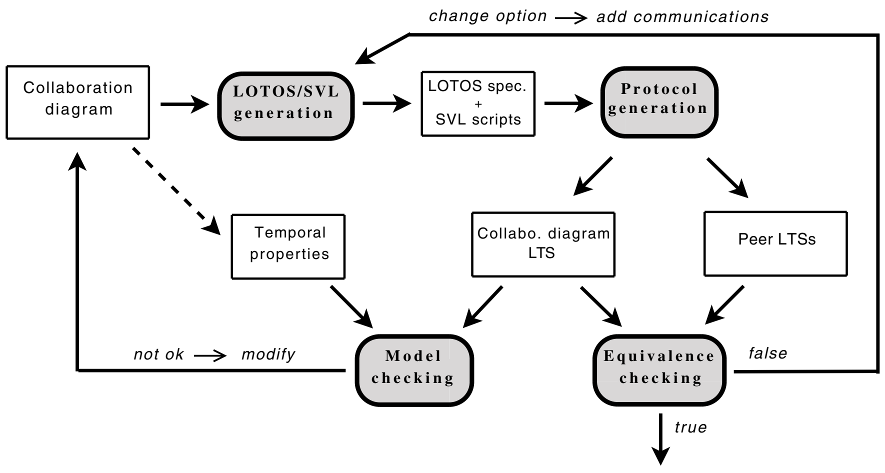


Fig. 6. Tool overview

by the choreography specification. Last, Bisimulator is used to check that the collaboration diagram LTS is equivalent to the distributed peer implementation.

Our approach, and especially the tool we implemented (cd2lotos), was applied and validated on about 85 collaboration diagrams (which resulted in the generation of about 49,000 lines of LOTOS and 23,000 lines of SVL). We also tested realizability on all these case studies, and all the unrealizable ones were checked equivalent once additional communications were inserted in the peer protocols.

Table 1 shows experimental results[4] on some of the examples belonging to our database. For each experiment, the table gives the size of the diagram in terms of number of peers, messages, and threads. Next, the table contains the number of lines of LOTOS and SVL generated by our prototype as well as the size (number of states and transitions) of the LTS generated from the diagram. Last, we give realizability results for both synchronous and asynchronous communication, and the time needed to compute both realizability checks. Example cd-045 corresponds to the case study presented in this paper.

First of all, it takes 1.9s to our prototype to generate SVL and LOTOS files for all the examples of our database for both communication models (synchronous and asynchronous) and both strategies (with and without additional communications). For medium size examples (cd-008, cd-025, cd-045), the generation of all intermediate LTSs and the realizability checks are quite fast (less than 20 seconds). For bigger diagrams (cd-059, cd-072), the computation time increases up to several minutes. It is interesting to note that examples involving more threads (cd-064) induce time consuming computations since they generate bigger intermediate state spaces due to the higher number of interleavings coming with the number of threads.

[4] Experiments have been carried out on a Vaio VGN-FZ11Z (Intel Core 2 Duo Processor T7300 2GHz, 2GB of RAM).

Table 1. Realizability results for some case studies (no additional communications)

Collab.	Size			LOTOS	SVL	CD LTS	Realizability		
diagrams	peers	messages	threads	(lines)	(lines)	(st./tr.)	sync.	async.	time
cd-008	5	9	4	388	148	27/46	√	√	19.56s
cd-025	4	6	3	304	130	12/15	√	√	16.20s
cd-045	5	8	3	341	130	10/13	√	×	18.69s
cd-059	10	20	3	666	238	56/175	×	×	1m12.31s
cd-064	7	13	6	495	184	96/396	×	×	1m46.14s
cd-072	16	30	4	959	346	220/748	×	×	6m31.39s

Table 2. Realizability results for some case studies (additional communications)

Collab.	Size			LOTOS	SVL	CD LTS	Realizability		
diagrams	peers	messages	threads	(lines)	(lines)	(st./tr.)	sync.	async.	time
cd-045	5	8	3	343	134	10/13	√	√	17.09s
cd-059	10	20	3	674	242	56/175	√	√	44.45s
cd-064	7	13	6	501	188	96/396	√	√	1m25.25s
cd-072	16	30	4	974	350	220/748	√	√	6m51.51s

Table 2 shows results obtained for the unrealizable examples presented in Table 1 once some additional communications are inserted. Obviously, all these examples are realized by adding these communications. Notice that realizability tests may take less time compared to Table 1 (cd-059, cd-064) because adding communications increases the sequentiality of the system, and therefore reduces communication interleaving.

During the experiments, we had to face the state explosion problem. In a first attempt, we were computing distributed systems in a single step, but, even for simple examples, the state space compilation lasted several minutes. Experiments showed that for collaboration diagrams of medium size (4/5 peers and 10/15 messages), the compilation of couples *(peer, queue)* was returning LTS containing hundreds even thousands of states (*resp.* transitions). Consequently, we decided to build first each couple *(peer, queue)*, minimize them, and compose them to finally obtain the expected system. This technique (known as compositional verification in CADP) allows to generate any step of the (distributed) system computation in less than one second.

5 Related Work

Several works aimed at studying and defining the realizability problem for choreography [10,3,11,6,2]. In [3,11], the authors define models for choreography and orchestration, and formalize a conformance relation between both models. These models are assumed given as input whereas we focus on the generation of one from the other (generation of peers from a global specification) while ensuring conformance. On a wider scale, all these approaches focus on theoretical aspects and most of them lack of tool support. WSAT [5] is the only tool we know which takes conversation protocols as input, and checks a set of realizability conditions on them. Our proposal is fully automated by tools. Moreover, these works only test realizability, but do not try to modify or extend peers to make them implementable as we do.

Other works [4,15] propose well-formedness rules to enforce the specification to be realizable. For example, in [4], the authors rely on a π-calculus-like language and session types to formally describe choreographies. Then, they identify three principles for global description under which they define a sound and complete end-point projection, that is the generation of distributed processes from the choreography description. We consider this solution too restricted since it may prevent the designer from specifying what (s)he wants to. In addition, it makes the choreography design more complicated since the designer cannot only focus on composition issues, but has to consider at the same time these well-formedness rules.

Last, to the best of our knowledge, the only work which proposes to add messages in order to implement unrealizable choreographies is [15]. To do so, the authors modify their choreography language to take new constructs (named dominated choice and loop) into account. During the projection of these new operators, some communications are added in order to make peers respect the choreography specification. This solution complicates the design because these new constructs are more restricting than the original ones, and they oblige the designer to explicit extra-constraints in the choreography specification by associating *dominant roles* to certain peers.

With respect to all these works, ours allows to implement any choreography specification (here written with collaboration diagrams) without adding any rule or constraint on the choreography language or specifications written with it. Furthermore, the LOTOS encoding makes possible the complete automation of realizability test, choreography verification, and peer generation (possibly with additional messages). Last but not least, we consider in this paper both synchronous and asynchronous communication models.

6 Concluding Remarks

In this paper, we have presented an encoding of collaboration diagrams into LOTOS in order to detect realizability issues, and if necessary solve them while generating peers by adding some communications. Our proposal deals with synchronous communication but also bounded asynchronous communication, and allows a completely automated and smooth process thanks to a prototype tool we implemented to generate LOTOS code, and the use of the CADP toolbox to analyze results generated from this code.

We have not discussed implementation issues here because it was out of scope. However, from peers generated using our approach either new services can be implemented, or some wrappers can be generated if an implementation of a service already exists [16]. In the latter case, the wrapper aims at constraining the functionality of the existing service to make it respect the application order of operations as specified in the generated peer behaviour. Implementation of executable services (Java, BPEL) from abstract descriptions can be achieved using Pi4SOA technologies [1], or following guidelines presented in [12].

As regards future works, a first perspective aims at extending our approach by considering as input to our problem a set of collaboration diagrams. Indeed, choice is a missing construct in the collaboration diagram notation, and using a set of diagrams allows to fill this gap. Second, realizability results for asynchronous communication were computed with various queue sizes. During these experiments, we noticed that results for queues of size one can be generalised to any size (*i.e.*, if a collaboration diagram is realizable for peers with queues of size one, it will be realizable too for queues of size k). Intuitively, this is because the equivalence check involves only sent messages, and received messages can be run at any time without any control. However, although this conjecture was experimentally validated, we would like to go forward and formally prove that realizability results for queues of size one hold for queues of size k and unbounded queues. Last, additional communications inserted in peer protocols to make them respect the collaboration diagram choreography can be minimized. In this paper, we systematically added a new message for each sequence of two actions in every thread, as well as for each dependency relation. However, all these messages are not always useful, and removing some of them for certain collaboration diagrams does not invalid their realizability. We would like to propose automatic techniques which figure out the minimal number of necessary additional messages to implement a given collaboration diagram.

Acknowledgements. The authors thank Javier Cámara and José Antonio Martín for fruitful discussions and interesting comments on a former version of this paper. This work has been partially supported by project TIN2008-05932 funded by the Spanish Ministry of Innovation and Science, project P06-TIC2250 funded by the Andalusian local Government, and US National Science Foundation Grants CCF-0614002 and CCF-0716095.

References

1. Pi4SOA Project, `http://www.pi4soa.org`
2. Bultan, T., Fu, X.: Specification of Realizable Service Conversations using Collaboration Diagrams. Service Oriented Computing and Applications 2(1), 27–39 (2008)
3. Busi, N., Gorrieri, R., Guidi, C., Lucchi, R., Zavattaro, G.: Choreography and Orchestration Conformance for System Design. In: Ciancarini, P., Wiklicky, H. (eds.) COORDINATION 2006. LNCS, vol. 4038, pp. 63–81. Springer, Heidelberg (2006)
4. Carbone, M., Honda, K., Yoshida, N.: Structured Communication-Centred Programming for Web Services. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 2–17. Springer, Heidelberg (2007)

5. 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)
6. Fu, X., Bultan, T., Su, J.: Synchronizability of Conversations among Web Services. IEEE Transactions on Software Engineering 31(12), 1042–1055 (2005)
7. Garavel, H., Lang, F.: SVL: A Scripting Language for Compositional Verification. In: Proc. of FORTE 2001, pp. 377–394. Kluwer, Dordrecht (2001)
8. Garavel, H., Mateescu, R., Lang, F., Serwe, W.: CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 158–163. Springer, Heidelberg (2007)
9. ISO. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. Technical Report 8807, International Standards Organisation (1989)
10. Kazhamiakin, R., Pistore, M.: Analysis of Realizability Conditions for Web Service Choreographies. In: Najm, E., Pradat-Peyre, J.-F., Donzeau-Gouge, V.V. (eds.) FORTE 2006. LNCS, vol. 4229, pp. 61–76. Springer, Heidelberg (2006)
11. Li, J., Zhu, H., Pu, G.: Conformance Validation between Choreography and Orchestration. In: Proc. of TASE 2007, pp. 473–482. IEEE Computer Society Press, Los Alamitos (2007)
12. Mateescu, R., Poizat, P., Salaün, G.: Adaptation of Service Protocols using Process Algebra and On-the-Fly Reduction Techniques. In: Bouguettaya, A., Krüger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 84–99. Springer, Heidelberg (2008)
13. Mateescu, R., Sighireanu, M.: Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus. Science of Computer Programming 46(3), 255–281 (2003)
14. Milner, R.: Communication and Concurrency. International Series in Computer Science. Prentice-Hall, Englewood Cliffs (1989)
15. Qiu, Z., Zhao, X., Cai, C., Yang, H.: Towards the Theoretical Foundation of Choreography. In: Proc. of WWW 2007, pp. 973–982. ACM Press, New York (2007)
16. Salaün, G.: Generation of Service Wrapper Protocols from Choreography Specifications. In: Proc. of SEFM 2008, pp. 313–322. IEEE Computer Society Press, Los Alamitos (2008)

Modelling Divergence in Relational Concurrent Refinement

Eerke Boiten[1] and John Derrick[2]

[1] Computing Laboratory, University of Kent, Canterbury, Kent, UK
E.A.Boiten@kent.ac.uk

[2] Department of Computer Science, University of Sheffield, Sheffield, UK
J.Derrick@dcs.shef.ac.uk

Abstract. An integration of state-based and behavioural formalisms can be obtained by imposing a concurrency semantics on a relational formalism. The data refinement theory for relational languages then provides a method for verifying the concurrent refinement relation. In this paper we investigate how divergence can be modelled relationally, and in particular show how differing process algebraic interpretations of divergence can be embedded in a relational framework. In doing so we derive relational simulation conditions for process algebraic refinement incorporating divergence.

Keywords: Data refinement, simulations, internal operations, process algebraic refinement preorders, divergence.

1 Introduction

The modelling and understanding of divergence is important in computer science. It plays an especially important role in refinement, where how divergence is modelled, and how it is treated in a development step lead to differences and subtleties. These distinctions are more prominent in a process algebra or behavioural setting where many refinement preorders have been defined, reflecting different choices of what is taken to be observable and, within these, different choices of divergence modelling.

For example, in a process algebra such as CSP [23,30] a system is defined in terms of actions (or events) which represent the interactions between a system and its environment. The differing semantics of CSP are denotational, associating one or more sets with each process, for example traces, refusals, divergences. Refinement is then defined in terms of set inclusions and equalities between the corresponding sets for different processes. Even without considering different models of divergence, many possible choices of semantics can be made, and a survey of many of these is given in [20,19]. Divergence in process algebraic models often arises from systems being able to engage in an infinite sequence of hidden (i.e., unobservable) actions. However, semantic models give a variety of interpretations to this situation and its consequences.

In a state-based system, e.g., one specified in Z, specifications are considered to define abstract data types (ADTs), consisting of an initialisation, a collection of operations and a finalisation. A program over an ADT is a sequential composition of these elements. Refinement is defined to be the subset relation over program behaviours for all possible programs, where what is deemed visible is the input/output relation. The accepted approach to make verification of refinements tractable is through downward and upward simulations which are sound and jointly complete [12]. Divergence in state-based models is almost universally taken to be due to partial definition, although the role of divergence due to internal events has also been considered [17].

In integrated notations, for both practical and theoretical reasons, it is important to keep track of how divergence arises, whether from the process algebraic or state-based angle, and how this impacts on refinement. Our ongoing research on "relational concurrent refinement" [14,4,16] contributes to this agenda, by explicitly recording in a relational setting the observations characterising process algebraic models. This allows the interpretation of relational formalisms like Z in a concurrency model, and also the verification of concurrent refinement through the standard relational method of simulations. In this work so far, we have only considered divergence in the context of the CSP-based failures-divergence semantics, with the specific "catastrophic" notion of divergence, see [4]. The current paper extends this by considering other published refinement relations for potentially diverging processes, in particular also ones which interpret divergence in a non-catastrophic way. In doing so, we uncover some of the properties and limitations of the relational concurrent refinement paradigm in general.

The structure of this paper is as follows. In Section 2 we discuss process algebraic refinement relations including how divergence can be modelled. In Section 3 we provide the basic definitions and background of the state-based, or relational, view of refinement, including our "relational concurrent refinement" approach, how CSP failures-divergences semantics is modelled in this, and some reflections on previous work. In Section 4 we model further refinement relations in our framework, providing simulation rules for a number of process algebraic preorders. The final section presents some conclusions, and areas for future work.

2 Process Algebraic Based Refinement

Process algebras [23,26,2] provide a means to describe and verify concurrent systems and processes. The semantics of a process algebra is either denotational (as in CSP) or else is given by means of a structural *operational* semantics which associates a labelled transition system (LTS) to each term; thus, its root is the state corresponding to the process term. Equivalences, and preorders, can be defined over the semantics where two terms are identified whenever no observer can notice any difference between their external behaviours. Such definitions are given in terms of a function O that represents the *set* of observations one could make while interacting with a process – i.e., generating a *denotational* semantics. For every such O we can define $p \sqsubseteq_O q$ iff $O(q) \subseteq O(p)$. For any preorder $\sqsubseteq_X$ we define: $p \equiv_X q$ iff $p \sqsubseteq_X q$ and $q \sqsubseteq_X p$. Thus, $p \equiv_O q$ iff $O(q) = O(p)$.

Varying how the environment *interacts* with a process leads to differing observations and thus different preorders (i.e., refinement relations). For systems without divergence an extensive overview of the choices is given in [18,20]; the successor paper [19] surveys refinement preorders in the presence of silent steps and the resultant possibility of divergence.

2.1 Notation

We assume here that the reader has a working knowledge of process algebraic refinement, we also assume the usual notation for labelled transition systems (LTSs) which are given in terms of a transition relation $T \subseteq \mathit{States} \times \mathit{Act} \times \mathit{States}$ over the states and the set of actions Act.

We use the usual notation for writing transitions as $p \xrightarrow{a} q$ for $(p, a, q) \in T$ and the extension of this to traces (written $p \xrightarrow{tr} q$), and to the presence of internal events (thus writing $p \overset{tr}{\Longrightarrow} q$). A sequence of events $\sigma \in \mathit{Act}^*$ is a trace of a process p if $\exists\, q \bullet p \overset{\sigma}{\Longrightarrow} q$. $\mathit{Tr}(p)$ denotes the set of traces of p. For a trace p and set of actions A, $p \restriction A$ denotes p with all actions outside A removed.

The set of initial actions of a process is defined as: $\mathit{next}(p) = \{a \in \mathit{Act} \mid \exists\, q \bullet p \overset{a}{\Longrightarrow} q\}$. p *after* σ is the set of all states reachable from p by performing the trace σ, i.e., $\{q \mid p \overset{\sigma}{\Longrightarrow} q\}$. $\mathit{Ref}(p, \sigma)$ is the refusal set of p after the trace σ, i.e., $\{X \subseteq \mathit{Act} \mid \exists\, q \bullet p \overset{\sigma}{\Longrightarrow} q \text{ and } X \cap \mathit{next}(q) = \varnothing\}$.

Throughout this paper we use i to denote the internal, or unobservable, event. The set of processes q allowing infinite internal evolution, denoted $q \xrightarrow{i^\infty}$, is the largest set of processes which allow an internal transition to another state in that set.

2.2 Refinement Relations Ignoring Divergence

A number of different effects can give rise to divergence in a system, and these can be modelled in a specification of a system. In the context of a process algebra, divergence is usually associated with the occurrence of *livelock*, whereby an unbounded amount of internal computation takes place.

A number of process algebraic refinement relations ignore divergence. Thus they provide the most non-catastrophic interpretation of divergence. These include observational equivalence from CCS [26] (which is the standard notions of equivalence between processes in CCS), trace refinement, reduction and testing equivalence.

Although these were not considered in [20], reduction and testing equivalence have been motivated [7,8] in the context of testing from LOTOS specifications [5,7]. Reduction (also called the testing preorder [11]) in the absence of divergence is identical to the failures preorder. The equivalence $\equiv_{red}$ induced by that preorder is also called testing equivalence. Leduc [25] documents the relationship between these and other relations in some detail. They can be defined as follows.

Definition 1 (Trace refinement, reduction)
Let p, q be LTSs. Then

$$\begin{array}{l} p \sqsubseteq_{tr} q \textit{ iff } Tr(q) \subseteq Tr(p) \\ p \sqsubseteq_{red} q \textit{ iff} \\ \qquad (a)\quad Tr(q) \subseteq Tr(p) \\ \qquad (b)\quad \forall \sigma : Tr(q) \bullet Ref(q,\sigma) \subseteq Ref(p,\sigma) \end{array} \qquad \Box$$

Thus $p \sqsubseteq_{red} q$ means that q has a subset of the traces of p, but deadlocks less often even in an environment whose traces are limited to those of q. Testing equivalence is obviously stronger than trace equivalence but weaker than weak bisimulation. Because divergences are ignored, both reduction and testing equivalence take the same non-catastrophic interpretation of divergence as observational equivalence.

2.3 Relations with Differing Interpretations of Divergence

We now turn to refinement relations that take divergence into account. Our first port of call in this discussion is the idea of divergent traces, with two basic notions as follows. First the catastrophic interpretation – that taken often in CSP. We define a predicate $p \uparrow \sigma$ to mean that there is a prefix of σ such that p may diverge after this prefix, and $Div_1(p)$ as the set of traces of p that have a divergent subtrace. Note that the notions of traces (Tr) and divergent traces (Div_1) are different from those in the denotational traces-divergences models for CSP: there divergences are upward closed, i.e. they consist of *all* strings in Act^* with a prefix in Div_1, and all of these are also included in the set of traces. This distinction explains some subtleties later on.

Definition 2 (Div_1 and $Conv_1$)

$$\begin{array}{l} p \uparrow \sigma \textit{ iff } \exists\, q, \sigma' \leq \sigma \bullet p \overset{\sigma'}{\Longrightarrow} q \xrightarrow{i^\infty} \\ p \downarrow \sigma \textit{ iff } \neg(p \uparrow \sigma) \textit{ i.e., iff } \forall\, \sigma' \leq \sigma \bullet \forall\, q \in p \textit{ after } \sigma' \bullet \neg(q \xrightarrow{i^\infty}) \\ \sigma \in Div_1(p) \textit{ iff } \sigma \in Tr(p) \wedge p \uparrow \sigma \\ \sigma \in Conv_1(p) \textit{ iff } \sigma \in Tr(p) \wedge p \downarrow \sigma \end{array} \qquad \Box$$

The sets Div_1 and $Conv_1$ have been used in the definition of a number of refinement relations, specifically in the definition of *may* and *must* testing [27,22,10]. Informally, p is may-refined by q if q may only do things that p may do; similarly for must-refinement, which by contraposition ("must allow" = "can't refuse") equals reverse failures refinement; t-refinement is their intersection.

Definition 3 (May, must and t-refinement)
Let p, q be LTSs. Then

$$\begin{array}{l} p \sqsubseteq_{may} q \textit{ iff } p \sqsubseteq_{tr} q \textit{ iff } Tr(q) \subseteq Tr(p) \\ p \sqsubseteq_{must} q \textit{ iff} \\ \qquad (a)\quad \forall\, \sigma \in Tr(p) \setminus Tr(q) \bullet q \uparrow \sigma \\ \qquad (b)\quad \forall\, \sigma \in Tr(p) \cap Conv_1(q) \bullet \sigma \in Conv_1(p) \wedge Ref(p,\sigma) \subseteq Ref(q,\sigma) \\ p \sqsubseteq_t q \textit{ iff } p \sqsubseteq_{may} q \wedge p \sqsubseteq_{must} q \end{array} \qquad \Box$$

One can illustrate the difference in approach in these relations via a number of simple examples (taken from [25]), where $\equiv_t$ is the equivalence induced by $\sqsubseteq_t$:

1. Given the behaviours defined by the diagrams in Figure 1 then we have $P \equiv_{red} Q$, but $\neg(P \sqsubseteq_{must} Q)$ and $\neg(P \sqsubseteq_t Q)$.
2. Given the behaviours defined by the diagrams in Figure 1 then we have $R \equiv_{red} S$, but $\neg(R \sqsubseteq_{must} S)$ and $\neg(R \sqsubseteq_t S)$.
3. Given the behaviours defined by the diagrams in Figure 2 then we have $\neg(U \equiv_{red} V)$, but $U \equiv_{must} V$ and $\neg(U \equiv_t V)$. The latter is caused by $\neg(U \equiv_{may} V)$.
4. Given the behaviours defined by the diagrams in Figure 2 then we have $\neg(X \sqsubseteq_{red} Y)$, but $X \equiv_{must} Y$ and $X \equiv_t Y$.

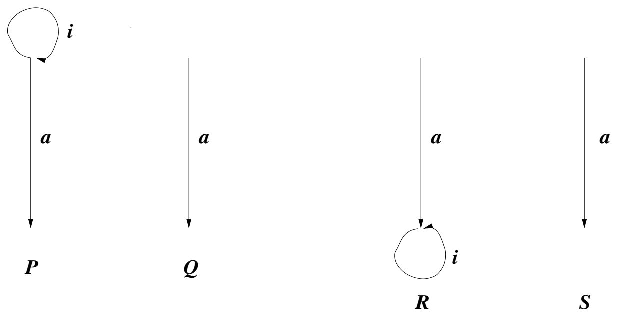


Fig. 1. Examples 1 and 2

As can be seen, and to summarise:

1. Some relations, e.g., testing equivalence, trace refinement, reduction, ignore divergence.
2. Others, such as $\equiv_t$, require that the sets of convergent and divergent traces are equal and refusal sets are only required to be equal for convergent traces.
3. Others still, such as $\equiv_{must}$ (i.e., failures-divergences equivalence), are less discriminating than $\equiv_t$. Here equivalent processes must have equal sets of convergent traces and refusal sets have to be equal for after convergent traces. However, they are just required to start diverging after the same traces - and after a divergent point they can have completely different behaviour.

In [25] Leduc argues that it makes more sense to use a more restrictive notion of divergence than that provided by the catastrophic interpretation. Specifically, he views a trace to be divergent if after that trace it is possible to engage in an infinite sequence of internal events. Such a non-catastrophic interpretation is also used in [1] (called a LTS with divergence whereby a predicate $\uparrow$ is defined on states in the LTS). This results in the following.

Definition 4 (*Div_2* and *$Conv_2$*)

$$\sigma \in Div_2(p) \text{ iff } \sigma \in Tr(p) \wedge \exists\, q \in p \text{ after } \sigma \bullet q \xrightarrow{i^\infty}$$
$$\sigma \in Conv_2(p) \text{ iff } \sigma \in Tr(p) \wedge \forall\, q \in p \text{ after } \sigma \bullet \neg(q \xrightarrow{i^\infty}) \qquad \square$$

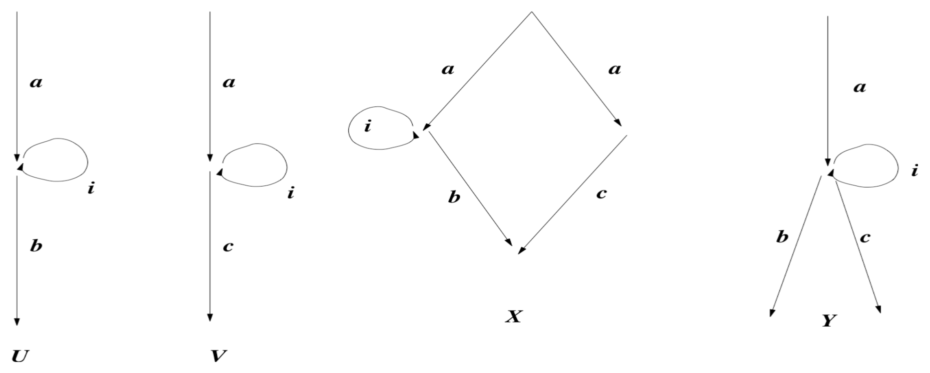


Fig. 2. Examples 3 and 4

Leduc argues that Div_2 and $Conv_2$ are more realistic in LTS models (as opposed to tree based models), and that in an LTS, divergence should be associated with the reachability of a divergent state, rather than the possibility of passing through a divergent state. Using these, further preorders and equivalences can be defined. For example, we can make the following definitions.

Definition 5 (*lte* refinement). *Let p, q be LTSs with alphabet Act. Then $p \sqsubseteq_{lte} q$ iff*

$$\begin{array}{ll}(a) & Tr(q) \subseteq Tr(p)\\ (b) & Div_2(q) \subseteq Div_2(p)\\ (c) & \forall \sigma \in Tr(q) \bullet Ref(q,\sigma) \subseteq Ref(p,\sigma)\end{array}$$ □

The intuition behind this is that $\equiv_{lte}$ requires equality of refusal sets even after points of divergence.

Similar ideas have been explored in a CSP context by Roscoe [31] following on from work on process algebra congruence relations by Puhakka and Valmari [28].

3 Relational Refinement

Having presented the process algebraic theory, we now turn to the refinement theory for abstract data types in a relational setting, see also [13]. The relational model of data refinement as described by He, Hoare and Sanders [21] initially required all operations to be total relations, but this unnecessary restriction was omitted in later presentations of the theory by these authors. The refinement theory of Z [32,13] is based on the original theory, embedding partial operations into the total ones. The standard way of doing so for Z, where domains get interpreted as preconditions, is called the *non-blocking model*. In previous work on relational concurrency semantics we have used an alternative embedding of partial relations into total relations, called the *blocking model*, which views the domain as a guard. We discuss the partial relations model in detail below, then briefly discuss the blocking model, and then describe how these both have been used in embedding concurrency semantics. We also discuss the issues that arise with the relational modelling of data types with internal operations.

3.1 A Partial Relational Model

A program (defined here as a finite sequence of operations) is given as a relation over a global state G, implemented using a local state State. A *data type* is a tuple $(\mathsf{State}, \mathsf{G}, \mathsf{Init}, \{\mathsf{Op}_k\}_{k \in Act}, \mathsf{Fin})$, where the operations $\{\mathsf{Op}_k\}$, indexed by $k \in Act$, are relations on the set State; Init is a total relation from G to State; Fin is a total relation from State to G.

A *complete program* over a *data type* $\mathsf{D} = (\mathsf{State}, \mathsf{G}, \mathsf{Init}, \{\mathsf{Op}_k\}_{k \in Act}, \mathsf{Fin})$ is an expression of the form $\mathsf{Init} \mathbin{⨾} \mathsf{P} \mathbin{⨾} \mathsf{Fin}$, where P, a relation over State, is a sequential composition of operations from $\{\mathsf{Op}_k\}_{k \in Act}$. For every sequence p over Act, and data type D, p_D denotes the complete program over D characterised by p.

Definition 6 (Data refinement for partial relations)
For partial data types A *and* C*,* C refines A*, denoted* $\mathsf{A} \sqsubseteq_{data} \mathsf{C}$*, iff for each finite sequence* p *over* Act*,* $\mathsf{p}_\mathsf{C} \subseteq \mathsf{p}_\mathsf{A}$. □

Downward and *upward* simulations form a sound and jointly complete [21,12] proof method for verifying refinements. In a simulation a step-by-step comparison is made of each operation in the data types, and to do so the concrete and abstract states are related by a retrieve relation.

Definition 7 (Downward simulation)
Assume data types $\mathsf{A} = (\mathsf{AState}, \mathsf{G}, \mathsf{AInit}, \{\mathsf{AOp}_k\}_{k \in Act}, \mathsf{AFin})$ *and* $\mathsf{C} = (\mathsf{CState}, \mathsf{G}, \mathsf{CInit}, \{\mathsf{COp}_k\}_{k \in Act}, \mathsf{CFin})$*. A* downward *simulation is a relation* R *from* AState *to* CState *satisfying*

$$\begin{array}{l}\mathsf{CInit} \subseteq \mathsf{AInit} \mathbin{⨾} \mathsf{R} \\ \mathsf{R} \mathbin{⨾} \mathsf{CFin} \subseteq \mathsf{AFin} \\ \forall\, k : Act \bullet \mathsf{R} \mathbin{⨾} \mathsf{COp}_k \subseteq \mathsf{AOp}_k \mathbin{⨾} \mathsf{R}\end{array}$$
□

Any relational data types A and C in this section are assumed to be defined as in the above definition; in later sections, *i-data types* A and C will have, in addition to this, an internal operation called i_A or i_C as an extra component.

Definition 8 (Upward simulation)
For data types A *and* C*, an* upward *simulation is a relation* T *from* CState *to* AState *such that*

$$\begin{array}{l}\mathsf{CInit} \mathbin{⨾} \mathsf{T} \subseteq \mathsf{AInit} \\ \mathsf{CFin} \subseteq \mathsf{T} \mathbin{⨾} \mathsf{AFin} \\ \forall\, k : Act \bullet \mathsf{COp}_k \mathbin{⨾} \mathsf{T} \subseteq \mathsf{T} \mathbin{⨾} \mathsf{AOp}_k\end{array}$$
□

If we interpret the "attempted" application of an operation outside its domain as a deadlock, this model observes deadlock by not returning a result for a particular program. However, if the *same* program allows another sequence of states where it does not deadlock, the original deadlock is unobservable (through relational union), and thus only *certain* deadlock is observed. The information contained in the observations of these data types is mostly determined by the choice of the global state G. In the simplest possible case, G is a singleton set $\{*\}$ and all that can be observed is a relation on G indicating whether a program must deadlock (the empty relation) or not (the maximal relation $\{(*,*)\}$).

3.2 The Blocking Model of Totalised Relations

Soundness and joint completeness of upward and downward simulations hold both when the operations are required to be total, and when they are not. In the latter case, also a *different* refinement theory can be obtained by first embedding the partial relations into total relations, and then applying the simulation rules. Two variants of this exist: the non-blocking model (traditionally used for Z, but of no further relevance to this paper) and the blocking model discussed here.

In the blocking model, a relational formalism like Z is given a semantics which corresponds more closely to a reactive than to a sequential model of computation. In particular, it views the domain of an operation as a guard which may not be weakened. This effect is encoded by *totalising* the relations in a particular way, turning a partial relation on a set S into a total relation on a set $\mathsf{S}_\perp$, which is S extended with a distinguished value $\perp$ not already in S or G. The value $\perp$ denotes the effect of applying an operation outside its domain; by explicitly representing this, this semantics observes possible as well as certain deadlock. For the simplest basic global state $\mathsf{G} = \{*\}$, possible program outcomes are subsets of $\mathsf{G}_\perp = \{*, \perp\}$: $\{\perp\}$ represents certain deadlock, $\{\perp, *\}$ possible deadlock, and $\{*\}$ absence of deadlock.

Characterisations of downward and upward simulations on these totalised relations can be simplified to remove any reference to $\perp$, see [14] for full details of the embedding, the derivation of the rules and the definition of upward simulation.

Definition 9 (Blocking downward simulation)
Given data types A *and* C *where the operations may be partial. A blocking* downward *simulation is a relation* R *from* AState *to* CState *satisfying*[1]

$$
\begin{array}{l}
\mathsf{CInit} \subseteq \mathsf{AInit} \mathbin{⨾} \mathsf{R} \\
\mathsf{R} \mathbin{⨾} \mathsf{CFin} \subseteq \mathsf{AFin} \\
\forall\, k : Act \bullet \operatorname{ran}(\operatorname{dom} \mathsf{AOp}_k \lhd \mathsf{R}) \subseteq \operatorname{dom} \mathsf{COp}_k \\
\forall\, k : Act \bullet \mathsf{R} \mathbin{⨾} \mathsf{COp}_k \subseteq \mathsf{AOp}_k \mathbin{⨾} \mathsf{R}
\end{array}
\qquad \square
$$

We have recently shown, however, that the upward and downward simulations given above are *not* jointly complete for blocking refinement [3], which means that completeness of any simulation rules derived from this needs to be proved separately. This contributes to making the blocking model less interesting as a computational model.

[1] $P \lhd R$ is the relation R constrained to the domain P, i.e., $\{(x, y) : R \mid x \in P\}$.

3.3 Relational Concurrent Refinement

Work by Bolton et al. [6] highlighted the subtle difference between blocking relational refinement and failures refinement. In response to this, we started a line of research exploring how concurrency semantics could be characterised by making extra observations in a relational formalism. This research programme of "Relational concurrent refinement" has three main objectives:

– to provide relational specification mechanisms (such as Z) with a concurrency semantics, e.g. to allow the integration of state-based and behavioural specification;
– to provide a way of verifying concurrency refinement inductively on an action-by-action basis (rather than e.g. computing inclusions of sets), through the relational method of simulations;
– to increase understanding of the relations between various concurrency semantics and refinement relations.

We first investigated in [14] how blocking refinement could be strengthened to failures refinement by including additional observable aspects. We showed that by observing refusal sets in finalisation, failures refinement could be recovered from blocking relational refinement. In a similar way, simulations for readiness refinement could be derived by observing ready sets at finalisation in the partial relational model. The paper [14] also discussed failures refinement for several semantic models for operations with inputs and outputs. In [4] internal operations were included in the discussion, and forms of relational data refinement were shown to be in correspondence with traces-divergences and failures-divergences refinement in a process semantics. The most recent research in this programme [16] moves away from a CSP-like context to consider how a variety of other process algebraic refinement relations can be defined in a state-based interpretation.

We can make a few observations and reflections on our previous work which turn out to be relevant for the refinement relations to be modelled in the rest of this paper. First, note we have used two refinement models for our embeddings: the *partial* relations model for readiness refinement, and the *blocking* model for various forms of failures refinement. These were conscious choices, though not always the only options. For readiness refinement, we observed that we could not use the blocking model because [14] "*it has included observations (i.e., $\perp_{\mathsf{G}}$) that are simply not observed via a finalisation that really only looks at enabled events*". A more general way of putting this is the following. Any refinement relation encoded in the *partial* relations model will necessarily be conjoined with *trace refinement*, as this is the refinement relation that underlies it [16]. If the modelled relation implies trace refinement (as readiness refinement does), this is not harmful; if it does not, the resulting refinement rules will necessarily be incomplete, checking for a stronger relation than intended. Similarly, any embedding based on the *blocking* model will incur a conjunction with *blocking data refinement*, which is only slightly weaker than singleton failure refinement [6,29]. Thus, the real reason that readiness refinement cannot be embedded in the blocking model is that it does not imply blocking data refinement.

On the other hand, we chose the blocking model for failures(-divergences) modelling, although these refinement relations *do* imply trace refinement. Indeed, [4, Lemma 3] shows that this representation is redundant, as a program returns "blocking" $\perp$ exactly when one of its prefixes observes the refusal of the next action of the program. Thus, for failures modelling, we have a choice of model, as failures refinement implies both trace refinement and blocking data refinement.

3.4 Modelling Internal Actions

In order to model data types *with* internal actions in a relational formalism which does not provide directly for these, the relations will have to be extended to account for any internal behaviour. In LTS terms, we need to encode every path $\overset{a}{\Longrightarrow}$ as a transition $\overset{a}{\rightarrow}$. Internal evolution will need to be accounted for between any two operations, after initialisation, and before finalisation. The first two would be necessary in an LTS setting, too (all states reachable through $\overset{\epsilon}{\Longrightarrow}$ become initial); the latter has no LTS analogue but reflects the richer semantic framework that relational observations at finalisation provide. Composing with arbitrary internal behaviour in all those places is usually correct, but complicates the formulas. *Usually* correct, because some semantics (e.g. stable failures) only allow observations in "stable" states, i.e., where no internal actions are enabled. This seems to imply that finalisations need to be pre-composed with *maximal* internal behaviour, and this is indeed what was done in [4].

We first described refinement with internal operations for Z [17] (also [13, Chapter 11]), although we did not derive these so-called "weak refinement" rules from a relational characterisation. Internal behaviour was added both before and after operations in the downward simulation rules, and also after initialisation. In the Z context, finalisation is not affected by internal behaviour when internal operations are precluded from producing output. Divergence was given a light touch, by only disallowing infinite internal behaviour in the concrete specification, along the lines of Butler's approach for B [9].

The first coverage of internal operations in the relational concurrent refinement framework was in [15]. However, the encoding given there was too discriminating, by observing blocking of an operation where it would have become applicable after internal evolution. It also did not restrict observations to stable states.

In [4], we provided an embedding which resolved both these issues, based on a generalised notion of data type that allows for both blocking and (catastrophic) divergence. In that embedding, initialisation is followed by finite internal evolution. Operations are both preceded and followed by finite internal evolution. Finalisation is restricted to non-divergent states, and preceded by maximal internal evolution. This embedding was proved to correctly represent failures-divergences refinement using the KIV interactive theorem prover. We also included several models for operations with outputs. Theorems on the closure of simulations under internal behaviour allow some simplification of the resulting simulation rules.

We observed that the embedding of [4] in the blocking model contains redundant explicit blocking information. Actually, basing the failures-divergences model on the partial relations model instead has several advantages. First, van Glabbeek [19] observes that when the testing model observes refusals, it is immaterial whether observations are restricted to stable states or not. Thus, the restriction on finalisation is unnecessary. Second, prepending internal behaviour to operations is necessary in the blocking model to avoid recording blocking ($\perp$). For example, a process which does a then i then b should not record $\perp$ for the trace ab just because b cannot be applied right after a. However, the partial relations model avoids this potential problem: the blocking of b right after a will not be visible, as its empty result will be hidden by the result of a (including i) followed by b. Thus, we only need to include internal evolution *after* operations.

4 Divergence Modelling

We now return to the process algebra based preorders discussed in Section 2. Generally speaking, embeddings of these in the relational framework will need to ensure the following points:

- they need to incorporate internal behaviour into the operations, initialisation and possibly finalisation;
- if they ignore divergence, there are no further requirements;
- otherwise, they need to ensure the correct observations are made when the final state records divergence;
- catastrophic interpretations need to generate arbitrary behaviour from the point of divergence onwards, and propagate this into all subsequent operations.

Following [4], we will record divergence using a special value ω which is assumed not to be included in any local or global state space. For any set S, let $\mathsf{S}_\omega = \mathsf{S} \cup \{\omega\}$.

First, the refinement relations that ignore divergence.

Trace refinement (and may-refinement). As discussed also in [16], trace refinement in the absence of internal operations "is" the partial relations model. Including also internal operations is relatively simple. For the same reasons as discussed in the context of failures-divergences refinement, we only need to include internal operations after initialisation and operation.

Definition 10 (Embedding trace refinement ignoring divergence) *An i-data type* $\mathsf{D} = (\mathsf{State}, \mathsf{G}, \mathsf{Init}, \{\mathsf{Op}_k\}_{k \in Act}, i, \mathsf{Fin})$ *is embedded as the data type* $\widehat{\mathsf{D}} = (\mathsf{State}, \mathsf{G}, \widehat{\mathsf{Init}}, \{\widehat{\mathsf{Op}_k}\}_{k \in Act}, \mathsf{Fin})$ *where*

$$\widehat{\mathsf{Op}} = \mathsf{Op} \operatorname{⨾} i^*$$
$$\widehat{\mathsf{Init}} = \mathsf{Init} \operatorname{⨾} i^* \qquad \square$$

If D only makes trivial observations, i.e., $\mathsf{G} = \{*\}$, then so does $\widehat{\mathsf{D}}$, and furthermore their traces are identical, i.e., for every sequence p over Act

$$p_{\widehat{\mathsf{D}}} \;=\; \bigcup\nolimits_{q \upharpoonright Act = p} q_{\mathsf{D}}$$

or equivalently (recall that a non-empty result indicates a trace being possible in this basic model):

$$p_{\widehat{\mathsf{D}}} \neq \varnothing \;\equiv\; \exists\, q \bullet q \upharpoonright Act = p \wedge q_{\mathsf{D}} \neq \varnothing$$

This can easily be proved by induction over the length of p. The simulation rules deriving from this are those of Definitions 7 and 8 with internal behaviour inserted after all occurrences of operations and initialisation. In the absence of (observed) divergence, joint completeness of the simulations follows from joint completeness of the partial relations simulations, plus the fact that the data type with internal operations is refinement equivalent to its embedding as in Definition 10, see also [13] for the latter point.

Note that this trace refinement relation is different from trace inclusion in the CSP failures-divergences model, as that does take divergence into account. An embedding for CSP trace refinement would be a simplification of the failures-divergences embedding, with a trivial observation at finalisation instead of refusals, as follows:

Definition 11 (Embedding trace refinement (CSP f-d model)) *An i-data type* $\mathsf{D} = (\mathsf{State}, \mathsf{G}, \mathsf{Init}, \{\mathsf{Op}_k\}_{k \in Act}, i, \mathsf{Fin})$ *is embedded as the data type* $\widehat{\mathsf{D}} = (\mathsf{State}_\omega, \mathsf{G}, \widehat{\mathsf{Init}}, \{\widehat{\mathsf{Op}_k}\}_{k \in Act}\}, \widehat{\mathsf{Fin}})$ *where*

$$\widehat{\mathsf{Init}} = \mathsf{Init} \mathbin{;} i^* \;\cup\; \textbf{if } \operatorname{divi} \mathsf{Init} \textbf{ then } \mathsf{G} \times \mathsf{State}_\omega$$
$$\widehat{\mathsf{Op}} = \mathsf{Op} \mathbin{;} i^* \;\cup\; \operatorname{div} \mathsf{Op} \times \mathsf{State}_\omega$$
$$\widehat{\mathsf{Fin}} = \mathsf{Fin} \;\cup\; \{\omega\} \times \mathsf{G}$$
$$\operatorname{div} \mathsf{Op} =_{def} \{s : \mathsf{State} \mid \exists\, s' : \mathsf{State} \bullet (s, s') \in \mathsf{Op} \wedge s' \xrightarrow{i^\infty}\}$$
$$\operatorname{divi} \mathsf{Init} =_{def} \exists\, s : \operatorname{ran} \mathsf{Init} \bullet s \xrightarrow{i^\infty}$$
□

The derivation of simulation rules is similar to those in [13, Section 3.3] or [4, Section 4.1], leading to the following definition.

Definition 12 (Simulations for trace refinement (CSP f-d model)) *A relation* R *between* AState *and* CState *is a downward simulation between i-data types* A *and* C *iff* $\forall\, k : Act$ *we have:*[2]

$$\textbf{if } \operatorname{divi} \mathsf{CInit} \textbf{ then } \operatorname{divi} \mathsf{AInit} \textbf{ else } \mathsf{CInit} \mathbin{;} i^*_{\mathsf{C}} \subseteq \mathsf{AInit} \mathbin{;} i^*_{\mathsf{A}} \mathbin{;} \mathsf{R}$$
$$\mathsf{R} \mathbin{;} \mathsf{CFin} \subseteq \mathsf{AFin}$$
$$(\operatorname{div} \mathsf{AOp}_k) \mathbin{⩤} \mathsf{R} \mathbin{;} \mathsf{COp}_k \mathbin{;} i^*_{\mathsf{C}} \subseteq \mathsf{AOp}_k \mathbin{;} i^*_{\mathsf{A}} \mathbin{;} \mathsf{R}$$
$$\operatorname{dom}(\mathsf{R} \rhd \operatorname{div} \mathsf{COp}_k) \subseteq \operatorname{div} \mathsf{AOp}_k$$

A relation T *between* CState *and* AState *is an upward simulation between i-data types* A *and* C *iff* $\forall\, k : Act$

$$\textbf{if } \operatorname{divi} \mathsf{CInit} \textbf{ then } \operatorname{divi} \mathsf{AInit} \textbf{ else } \mathsf{CInit} \mathbin{;} i^*_{\mathsf{C}} \mathbin{;} \mathsf{T} \subseteq \mathsf{AInit} \mathbin{;} i^*_{\mathsf{A}}$$
$$\mathsf{CFin} \subseteq \mathsf{T} \mathbin{;} \mathsf{AFin}$$
$$\operatorname{dom}(\mathsf{T} \rhd \operatorname{div} \mathsf{AOp}_k) \mathbin{⩤} \mathsf{COp}_k \mathbin{;} i^*_{\mathsf{C}} \mathbin{;} \mathsf{T} \subseteq \mathsf{T} \mathbin{;} \mathsf{AOp}_k \mathbin{;} i^*_{\mathsf{A}}$$
$$\operatorname{div} \mathsf{COp}_k \subseteq \operatorname{dom}(\mathsf{T} \rhd \operatorname{div} \mathsf{AOp}_k)$$
□

[2] The relation $P \mathbin{⩤} R$ is R with P excluded from its domain, i.e., $\{(x, y) : R \mid x \notin P\}$.

Reduction. The embedding for reduction is a simplification of that for failures-divergences refinement, e.g. as given in [4], introducing an extra component E recording refused events, but removing the case distinctions and special treatment arising from infinite internal evolution.

Definition 13 (Embedding reduction). *An i-data type* $\mathsf{D} = (\mathsf{State}, \mathsf{G}, \mathsf{Init}, \{\mathsf{Op}_k\}_{k \in Act}, i, \mathsf{Fin})$ *is embedded as the data type* $\widehat{\mathsf{D}} = (\mathsf{State}, \mathsf{G} \times \mathbb{P}\, Act, \widehat{\mathsf{Init}}, \{\widehat{\mathsf{Op}_k}\}_{k \in Act}, \widehat{\mathsf{Fin}})$ *where*

$$\begin{aligned}
&\widehat{\mathsf{Init}} = \{((g, E), s) : (\mathsf{G} \times \mathbb{P}\, Act) \times \mathsf{State} \mid (g, s) \in \mathsf{Init} \mathbin{⨾} i^*\} \\
&\widehat{\mathsf{Op}} = \mathsf{Op} \mathbin{⨾} i^* \\
&\widehat{\mathsf{Fin}} = \{(s, (g, E)) : \mathsf{State} \times (\mathsf{G} \times \mathbb{P}\, Act) \mid \\
&\qquad (s, g) \in \mathsf{Fin} \wedge \forall\, k : E \bullet s \notin \mathrm{dom}(i^* \mathbin{⨾} \mathsf{Op}_k)\}
\end{aligned}$$

□

Note that the change to initialisation is only to account for the extra component E in the global state. The resulting simulation rules are identical to those for failures refinement with internal evolution added after all operations and initialisation, and before operations in precondition (refusal) computation. The multiple components observed in finalisation imply that the simulations are not in general complete: the simulations as given impose separate conditions on each component, whereas due to dependencies between the components weaker conditions may suffice. However, for *trivial* original finalisations, due to the same normal form argument as given for trace refinement above, these rules inherit the joint completeness of the failures refinement rules proved e.g. by Josephs [24].

Must-refinement and t-refinement. Although the characterisation given in Definition 3 is different (using a set of traces which does not contain the upward closure of divergences), this relation is identical to failures-divergences refinement in CSP, but reversing the direction, see also [19]. The crucial observation in this respect is that the conditions above impose no restrictions on traces with a divergent prefix in q, making all specified behaviour after divergence irrelevant.

Thus, must-refinement can be checked by using the failures-divergences simulations [4] in reverse order, i.e., swapping abstract and concrete. For the CSP traces model, failures-divergences refinement implies trace refinement; however, reverse must-refinement does not imply may-refinement due to may-refinement ignoring divergence.

We do not believe we can sensibly model t-refinement through an embedding, i.e., even if we could find a denotational model using sets where inclusion represents t-refinement, we do not believe this could lead to a complete simulation method. This is because any (e.g. downward) simulation relations derived from this would need to establish *both* a failure-divergence refinement (and thus CSP style trace refinement) in one direction, *and* a (divergence-ignoring) trace refinement in the opposite direction.

Non-catastrophic divergence. In a non-catastrophic interpretation, divergence is a property only of the state (whether it admits infinite internal evolution) and not of the trace (whether it may have come through such a state). Thus, embeddings for associated refinement relations are significantly simpler, not having to propagate divergence from one state to the next, nor having to introduce arbitrary behaviour in such states.

In the global state for an embedding of *lte*-refinement we need to record three components: the original global state; refusal information for the final state; and whether the final state was divergent. Also, we need to ensure that "non-divergent" is a subset of "divergent", so we will record the former by returning $\varnothing$ and the latter by returning both $\varnothing$ and $\{\omega\}$.

Definition 14 (Embedding *lte*-refinement). *An i-data type* $\mathsf{D} = (\mathsf{State}, \mathsf{G}, \mathsf{Init}, \{\mathsf{Op}_k\}_{k \in Act}, i, \mathsf{Fin})$ *is embedded as the data type* $\widehat{\mathsf{D}} = (\mathsf{State}, \mathsf{G} \times \mathbb{P}\, Act \times \mathbb{P}\{\omega\}, \widehat{\mathsf{Init}},$
$\{\widehat{\mathsf{Op}_k}\}_{k \in Act}, \widehat{\mathsf{Fin}})$ *where*

$$\begin{aligned}
&\widehat{\mathsf{Init}} = \{((g, E, d), s) : (\mathsf{G} \times \mathbb{P}\, Act \times \mathbb{P}\{\omega\}) \times \mathsf{State} \mid (g, s) \in \mathsf{Init} \,{}_9^\circ\, i^*\} \\
&\widehat{\mathsf{Op}} = \mathsf{Op} \,{}_9^\circ\, i^* \\
&\widehat{\mathsf{Fin}} = \{(s, (g, E, d)) : \mathsf{State} \times (\mathsf{G} \times \mathbb{P}\, Act \times \mathbb{P}\{\omega\}) \mid \\
&\qquad (s, g) \in \mathsf{Fin} \wedge (d = \varnothing \vee s \xrightarrow{i^\infty}) \wedge \forall\, k : E \bullet s \notin \operatorname{dom}(i^* \,{}_9^\circ\, \mathsf{Op}_k)\}
\end{aligned} \qquad \square$$

Note that this embedding includes observations of divergence in the global state only, as opposed to others which include it in the local state, or in both.

Definition 15 (Simulations for *lte*-refinement). *A relation* R *between* AState *and* CState *is a downward simulation between i-data types* A *and* C *iff* $\forall\, k : Act$

$$\begin{aligned}
&\mathsf{CInit} \,{}_9^\circ\, i_{\mathsf{C}}^* \subseteq \mathsf{AInit} \,{}_9^\circ\, i_{\mathsf{A}}^* \,{}_9^\circ\, \mathsf{R} \\
&\mathsf{R} \,{}_9^\circ\, \mathsf{CFin} \subseteq \mathsf{AFin} \\
&\operatorname{ran}(\operatorname{dom}(i_{\mathsf{A}}^* \,{}_9^\circ\, \mathsf{AOp}_k) \lhd \mathsf{R}) \subseteq \operatorname{dom}(i_{\mathsf{C}}^* \,{}_9^\circ\, \mathsf{COp}_k) \\
&\mathsf{R} \rhd \mathsf{C}\uparrow \subseteq \mathsf{A}\uparrow \\
&\mathsf{R} \,{}_9^\circ\, \mathsf{COp}_k \,{}_9^\circ\, i_{\mathsf{C}}^* \subseteq \mathsf{AOp}_k \,{}_9^\circ\, i_{\mathsf{A}}^* \,{}_9^\circ\, \mathsf{R}
\end{aligned}$$

where

$$\mathsf{State}\uparrow \;=_{def}\; \{s : \mathsf{State} \mid s \xrightarrow{i^\infty}\}$$

A relation T *between* CState *and* AState *is an upward simulation between i-data types* A *and* C *iff* $\forall\, k : Act$

$$\begin{aligned}
&\mathsf{CInit} \,{}_9^\circ\, i_{\mathsf{C}}^* \,{}_9^\circ\, \mathsf{T} \subseteq \mathsf{AInit} \,{}_9^\circ\, i_{\mathsf{A}}^* \\
&\mathsf{CFin} \subseteq \mathsf{T} \,{}_9^\circ\, \mathsf{AFin} \\
&\forall\, c : \mathsf{CState} \bullet \exists\, a : \mathsf{AState} \bullet (c, a) \in \mathsf{T} \wedge \\
&\qquad\qquad \forall\, k : Act \bullet a \in \operatorname{dom}(i_{\mathsf{A}}^* \,{}_9^\circ\, \mathsf{AOp}_k) \Rightarrow c \in \operatorname{dom}(i_{\mathsf{C}}^* \,{}_9^\circ\, \mathsf{COp}_k) \\
&\mathsf{C}\uparrow \subseteq \mathsf{T} \rhd \mathsf{A}\uparrow \\
&\mathsf{COp}_k \,{}_9^\circ\, i_{\mathsf{C}}^* \,{}_9^\circ\, T \subseteq \mathsf{T} \,{}_9^\circ\, \mathsf{AOp}_k \,{}_9^\circ\, i_{\mathsf{A}}^*
\end{aligned} \qquad \square$$

The conditions are the familiar ones of failures refinement (bringing in the condition on domains from the blocking model in the downward simulation, and a stronger condition on domains in upward simulation), plus conditions on divergent states. As we have separated, in deriving these simulation rules, conditions between tuples (g, E, d) (where g is arbitrary) into conditions on their elements, completeness is not guaranteed.

5 Conclusions

In this paper we have discussed how divergence can be modelled relationally in the context of defining refinement relations. Specifically, we have shown how various interpretations of divergence as found in the process algebra literature can be embedded in a relational model. In doing so we have derived simulations for relational embeddings of a number of refinement preorders found in process algebras. Alongside successful embeddings, we have also highlighted a number of limitations of the relational concurrent refinement method, at least at present. Preorders which do not imply trace refinement, few as they may be, do not fit in well. Also, from embeddings of individual preorders we can construct an embedding for their intersection, but it seems unlikely that the resulting simulation rules would be complete in general.

Our earlier embedding of failures-divergences refinement in [4] was based on an intermediate data type, where each operation was split into three parts: the normal behaviour, its domain of blocking, and its domain of (catastrophic) divergence, together partitioning the state. The commonalities between the embeddings in this paper suggest further enhancements: removing the partitioning requirement in order to use the partial relations model, to allow for both explicit and implicit modelling of blocking; and addition of "transient error" behaviour such as the non-catastrophic divergence modelled in this paper. (Both the blocking and divergent behaviour essentially make all operations after the point of "error" irrelevant.) Further alternative preorders e.g. relating to "unfair testing equivalence" (identifying blocking and divergence) [25] could be modelled using such an intermediate type, too. It would also be interesting to see how Roscoe's denotational semantics "looking beyond divergence" [31] fits in with these preorders.

This paper only addressed divergence due to internal events, which in an operational view may also be taken to include unguarded recursion. Divergence due to partial definition is the norm in relational formalisms, and indeed the essence of the non-blocking model referred to in Section 3. Leduc also discusses divergence due to a lack of image finiteness. There the intuition is that the mechanism for resolving an infinitely branching internal choice may consume infinite time. This could also be modelled using the intermediate data type of [4] which abstracts from the particular cause of divergence.

Further work also includes mechanisation of this theory (along the lines found in [4]) and the unification of the refinement relations as defined for IO automata.

References

1. Abramsky, S.: Observation equivalence as a testing equivalence. Theor. Comput. Sci. 53(2-3), 225–241 (1987)
2. Bergstra, J.A., Ponse, A., Smolka, S.A. (eds.): Handbook of Process Algebra. Elsevier Science Inc., New York (2001)
3. Boiten, E.A., Derrick, J.: Incompleteness of relational simulations in the blocking paradigm (submitted for publication, 2008)
4. Boiten, E.A., Derrick, J., Schellhorn, G.: Relational concurrent refinement II: Internal operations and outputs. Formal Aspects of Computing (accepted for publication)
5. Bolognesi, T., Brinksma, E.: Introduction to the ISO Specification Language LOTOS. Computer Networks and ISDN Systems 14(1), 25–59 (1988)
6. Bolton, C., Davies, J.: Refinement in Object-Z and CSP. In: Butler, M., Petre, L., Sere, K. (eds.) IFM 2002. LNCS, vol. 2335, pp. 225–244. Springer, Heidelberg (2002)
7. Brinksma, E., Scollo, G.: Formal notions of implementation and conformance in LOTOS. Technical Report INF-86-13, Dept of Informatics, Twente University of Technology (1986)
8. Brinksma, E., Scollo, G., Steenbergen, C.: Process specification, their implementation and their tests. In: Sarikaya, B., Bochmann, G.v. (eds.) Protocol Specification, Testing and Verification, VI, Montreal, Canada, jun 1986, pp. 349–360. North-Holland, Amsterdam (1986)
9. Butler, M.: An approach to the design of distributed systems with B AMN. In: Till, D., Bowen, J.P., Hinchey, M.G. (eds.) ZUM 1997. LNCS, vol. 1212, pp. 223–241. Springer, Heidelberg (1997)
10. Cleaveland, R., Hennessy, M.: Testing equivalence as a bisimulation equivalence. In: Proceedings of the international workshop on Automatic verification methods for finite state systems, pp. 11–23. Springer, New York (1990)
11. de Nicola, R.: Extensional equivalences for transition systems. Acta Informatica 24(2), 211–237 (1987)
12. de Roever, W.-P., Engelhardt, K.: Data Refinement: Model-Oriented Proof Methods and their Comparison. Cambridge University Press, Cambridge (1998)
13. Derrick, J., Boiten, E.A.: Refinement in Z and Object-Z. Springer, Heidelberg (2001)
14. Derrick, J., Boiten, E.A.: Relational concurrent refinement. Formal Aspects of Computing 15(1), 182–214 (2003)
15. Derrick, J., Boiten, E.A.: Relational concurrent refinement with internal operations. In: Aichernig, B., Boiten, E.A., Derrick, J., Groves, L. (eds.) BCS-FACS Refinement Workshop. ENTCS, vol. 187, pp. 35–53 (2006)
16. Derrick, J., Boiten, E.A.: More relational concurrent refinement: traces and partial relations. In: Boiten, E.A., Derrick, J., Schellhorn, G. (eds.) Proceedings REFINE. ENTCS (to appear, 2008)
17. Derrick, J., Boiten, E.A., Bowman, H., Steen, M.: Specifying and Refining Internal Operations in Z. Formal Aspects of Computing 10, 125–159 (1998)
18. van Glabbeek, R.J.: The linear time - branching time spectrum. In: Baeten, J.C.M., Klop, J.W. (eds.) CONCUR 1990. LNCS, vol. 458, pp. 278–297. Springer, Heidelberg (1990)
19. van Glabbeek, R.J.: The linear time – branching time spectrum II; the semantics of sequential systems with silent moves (extended abstract). In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 66–81. Springer, Heidelberg (1993)

20. van Glabbeek, R.J.: The linear time - branching time spectrum I. The semantics of concrete sequential processes. In: Bergstra, J.A., Ponse, A., Smolka, S.A. (eds.) Handbook of Process Algebra, pp. 3–99. North-Holland, Amsterdam (2001)
21. Jifeng, H., Hoare, C.A.R., Sanders, J.W.: Data refinement refined. In: Robinet, B., Wilhelm, R. (eds.) ESOP 1986. LNCS, vol. 213, pp. 187–196. Springer, Heidelberg (1986)
22. Hennessy, M.: Algebraic theory of processes. MIT Press, Cambridge (1988)
23. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
24. Josephs, M.B.: A state-based approach to communicating processes. Distributed Computing 3, 9–18 (1988)
25. Leduc, G.: On the Role of Implementation Relations in the Design of Distributed Systems using LOTOS. PhD thesis, University of Liège, Liège, Belgium (June 1991)
26. Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)
27. De Nicola, R., Hennessy, M.: Testing equivalence for processes. In: Díaz, J. (ed.) ICALP 1983. LNCS, vol. 154, pp. 548–560. Springer, Heidelberg (1983)
28. Puhakka, A., Valmari, A.: Weakest-congruence results for livelock-preserving equivalences. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, pp. 510–524. Springer, Heidelberg (1999)
29. Reeves, S., Streader, D.: Data refinement and singleton failures refinement are not equivalent. Formal Aspects of Computing 20(3), 295–301 (2008)
30. Roscoe, A.W.: The Theory and Practice of Concurrency. International Series in Computer Science. Prentice-Hall, Englewood Cliffs (1998)
31. Roscoe, A.W.: Seeing beyond divergence. In: Abdallah, A.E., Jones, C.B., Sanders, J.W. (eds.) Communicating Sequential Processes. LNCS, vol. 3525, pp. 15–35. Springer, Heidelberg (2005)
32. Woodcock, J.C.P., Davies, J.: Using Z: Specification, Refinement, and Proof. Prentice-Hall, Englewood Cliffs (1996)

SAL-Based Symbolic Scheduling in Time-Triggered Networks

Sebastian Voss[1], Maria Sorea[1], and Klaus Echtle[2]

[1] EADS Innovation Works, Munich, Germany
{sebastian.voss,maria.sorea}@eads.net
[2] University of Duisburg-Essen, Essen, Germany
echtle@dc.uni-due.de

Abstract. This paper presents a novel approach for combined task and message scheduling for TDM-based avionics applications that allows to automatically compute schedules with minimal end-to-end latency. Our approach relies on a symbolic encoding of the scheduling problem. The symbolic encoding is done by constructing gradually a schedule beginning in a state where no task has started and no messages have been sent on the bus yet, and proceeding one step at a time assigning starting times to tasks and slot positions to messages. We use the SAL toolset from SRI for our experiments. Experimental results demonstrate how the latest generation of model-checking tools meet the challenges of providing both a convenient modeling language and the performance to solve given scheduling problems.

1 Introduction

Embedded systems in aerospace become more and more integrated in order to reduce weight, volume/size, and power of hardware for more fuel-efficiency. The trend is towards integrated modular avionics (IMA) architectures in which several functions share a common (fault tolerant) computing resource, and operate in a more integrated (i.e., mutually interactive) manner. Since the IMA approach allows multiple applications of different criticality levels to share common computing resources, it is important to keep individual application away from potential interference. The main way for protecting integrated applications and system resources is via temporal and spatial partitioning. Spatial partitioning guarantees that an application has exclusive control over its own data and state information. With spatial partitioning, an application can be protected from any erroneous behaviors of other applications while sharing same physical resources. Temporal partitioning guarantees that an application or communication server has temporal exclusive access to its pre-allocated resources. With guaranteed pre-scheduled temporal partitioning, applications can meet their timing requirements.

For enforcing temporal partitioning, shared resources have to be scheduled while guaranteeing timing constraints of the application. When considering distributed systems, one has to take into account not only the constraints imposed by the applications but also the characteristics and efficient usage of the underlying communication bus [1,2]. Therefore, an effective scheduling policy for TDM-based avionics applications has to consider both task and message scheduling.

We present a novel technique for task and message scheduling for TDM-based avionics applications that allows to automatically compute schedules with minimal end-to-end latency. Our approach relies on a symbolic encoding of the given scheduling problem [3]. This symbolic encoding is done by constructing gradually a schedule beginning in a state where no task has started and no messages have been sent on the bus yet, and proceeding one step at a time assigning starting times to tasks and slot positions to messages. Therefore, a symbolic state represents tasks that have not started, tasks that are running, and tasks that have finished. Symbolic model checking is employed to automatically search for a schedule with minimal end-to-end latency.

Experimental results demonstrate how the latest generation of model-checking tools (we use SAL from SRI) meets the challenges of providing both a convenient modeling language and the performance to solve given scheduling problems. These results show with respect to system size and the degree of intertask communication a limitation of system with up to 15 tasks caused by the given state-space explosion.

Combined task and message scheduling is considered by [4,5,1,6]. The approach advocated by Pree et al. addresses both task and message scheduling. However, different techniques are employed for task and message scheduling, namely, Earliest Deadline First (EDF) for task scheduling and an heuristic algorithm, adapted from Reverse EDF for message scheduling, respectively. Tindell and Clark [5] provide a holistic scheduling technique. Based upon a distributed real-time system where fixed-priority tasks with arbitrary deadlines communicate by message passing with a simple TDMA protocol, a window-based analysis technique is provided to calculate the worst-case response times of the distributed task set. Pop, Eles and Peng [1] provide an approach to find for a minimal worst case delay by which the system completes execution. A priority based schema is used to decide which processes or tasks are extracted in order to be scheduled at a given time. The algorithm provided by Abdelzaher and Kang [6] uses a branch-and-bound technique to find a task schedule. The algorithm takes into account delays, and precedence relations imposed by interprocess communications, and considers many possibilities for improving the scheduling lateness at the cost of complexity.

Symbolic task scheduling is considered by Jensen, Lauritzen and Laursen [7]. The authors use BDDs for representing the task graph scheduling problem with uniform processors and arbitrary task execution times. A breadth-first search and an A*-based algorithm is employed for finding optimal schedules. The representation of tasks is similar to this paper, however, they do not consider underlying network communication aspects, and message scheduling.

A SAT-based approach to task and message allocation problem of distributed real-time systems is considered by Metzner, Fränzle, Herde and Stierand [8]. An optimal strategy to assign task and messages to ECUs and network busses is done by modeling timing and resource restrictions as a set of integer formulae.

The paper is structured as follows. Section 2 formalizes the task and message scheduling problem in TDM-based networks. Section 3 introduce briefly the SAL framework, which is used for encoding the scheduling problem. Section 4 gives an overview over the employed analysis techniques and presents and discusses experimental results. We conclude in Section 5.

2 Scheduling in Time-Triggered Systems

Communication in time-triggered networks is realized by a time-division multiple-access (TDMA) discipline, in which n nodes share a time-triggered bus based on a cyclic schedule. Each node n_i can transmit only during a predetermined time interval, denoted as TDMA (or time) slot. A sequence of slots builds a TDMA round. Several rounds are combined into a cycle that is repeated periodically. Tasks are executed on nodes, and communicate with each other by message passing. A node might send messages (from a given set of messsages M) in several slots in a TDMA round.

In the following, the set of time slots is denoted by $SL = \{sl_1, \ldots, sl_k\}$. The time intervals associated to every slot sl_i are disjoint. A function $\sigma : M \rightarrow SL$ allocates slots to messages. The task and message scheduling technique presented here computes, for a given set of tasks T, (1) the starting time s_i for all tasks $t_i \in T$, and (2) the slot allocation function σ, that is the slot (or slots, in case several messages are sent) in which a task t_i is scheduled to transmit its messages on the communication bus. Formally, the task/message scheduling problem is defined as follows.

Definition 1. Let $N = \{n_1, n_2, ..., n_m\}$ be a set of nodes, $T = \{t_1, t_2, ..., t_n\}$ a set of tasks, and $M = \{m_1, m_2, ..., m_o\}$ a set of input/output messages of the tasks. The dependencies between the tasks in T are captured by a precedence graph $\mathcal{G}$. Furthermore, let $\eta : N \rightarrow T$ be a function that assigns to every node a task running on it, and $\tau : T \rightarrow 2^M$ a function that assigns a set of messages to tasks. The scheduling problem consists of determining the starting time and slot(s) position of the tasks in T that is, to calculate for every task $t_i \in T$ the tuple $\gamma_i = \langle s_i, \{\sigma(m_1), \ldots, \sigma(m_j)\}\rangle$, where s_i is the execution starting time of t_i, and $\sigma(m_1), \ldots, \sigma(m_j)$ the slots in which the messages $\{m_1, \ldots, m_j\}$ are transmitted. Furthermore, we define the *destination* of a task $t \in T$ under $m \in M$, as the function $\mathcal{D} : T \times M \rightarrow T$, with $\mathcal{D}(t, m) = \{t' \in T | \langle t, m, t'\rangle \in E\}$.

Remark 1. We consider the period P of a task to be equivalent to the communication cycle, thus identical for all tasks in T. Hence, the arrival times a_i, which denote the earliest time a task can be invoked is equivalent to the beginning of each communication cycle for all tasks. The deadline d_i, which is the latest time it can finish its execution corresponds to the end of the communication cycle. Preemption of tasks is not considered.

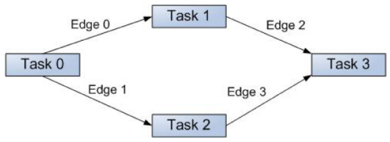


Fig. 1. Simple example of a Precedence Graph

Example 1. Consider a set of tasks $T = \{t_0, t_1, t_2, t_3\}$*, and a set of messages* $M = \{m_0, m_1, m_2, m_3\}$*, cf. figure 1. Furthermore,* $\tau(t_0) = \{m_0, m_1\}$*,* $\tau(t_1) = \{m_2\}$*,* $\tau(t_2) = \{m_3\}$*,* $\tau(t_3) = \{\}$*,* $\eta(n_0) = t_0$*,* $\eta(n_1) = \{t_1, t_2\}$ *and* $\eta(n_2) = \{t_3\}$*. For simplicity reasons we just use time units, and assume an equal computation time for each task with* $c_i = 2$ *time units. Each edge* $Edge_i$ *in figure 1 complies to the corresponding message* m_i *(*$Edge_0 \hat{=} m_0$*). The destination function is given as* $\mathcal{D}(t_0, m_0) = t_1$*,* $\mathcal{D}(t_0, m_1) = t_2$*,* $\mathcal{D}(t_2, m_3) = t_3$*,* $\mathcal{D}(t_1, m_2) = t_3$*.*

3 Translation to SAL

SAL (Symbolic Analysis Laboratory, `http://sal.csl.sri.com`) [9] is a framework for specification and analysis of concurrent systems. It consists of the SAL language, which provides notations for specifying state machines and their properties, and the SAL system that provides model checkers and other tools for analyzing properties of state machines specifications. The SAL model checkers are of interest for our analysis: the symbolic model checker sal-smc that uses the CUDD BDD package and provides access to many options for variable ordering, and for clustering and partitioning the transition relation, and the bounded model checking of infinite-state systems. This model checker relies on a solver for deciding the validity of logical formulas that can mix linear arithmetic, equalities with uninterpreted function symbols, and propositional logic. The default solver used by SAL is Yices [10]. In this way we can encode our scheduling problem both in discrete time and continuous time, and use either the symbolic model checker or the infinite-state model checker for computing our schedules.

We propose to solve the scheduling protocol as formulated in Definition 1 by using symbolic model checking. For this we need to encode this problem as a transition system. This is done by constructing gradually a schedule, following the precedence graph, beginning in a state where no task has started and no messages have been sent on the bus yet, and proceeding on step at a time assigning starting times to tasks and slots positions to messages. Our goal is to use SALs model checkers to calculate an integrated task and message schedule that is minimal in terms of end-to-end latency.

3.1 Precedence Graph

The given precedence graph, as specified in section 2, comprises several elements: a set of tasks, a set of messages and a set of nodes. Thus, we begin by defining the type declaration for these precedence graph elements in SAL. All given tasks are defined via the enumeration type `TASKS`. Furthermore, the enumeration types `MESSAGES` and `NODES` specify, respectively, all messages and all nodes of the given precedence graph.

```
TASKS : TYPE = {Task0, Task1, Task2, Task3};
MESSAGES : TYPE = {Edge0, Edge1, Edge2, Edge3};
NODES : TYPE = {Node0, Node1, Node2};
```

To specify tasks, messages and nodes, several parameters are defined in dedicated data structures, in particular, a task, a message, and a node record.

Task Record	Message Record	Node Record
`taskcalendar:TYPE=[#`	`msgcalendar:TYPE=[#`	`nodecalendar:TYPE=[#`
`t_started:BOOLEAN,`	`m_started:BOOLEAN,`	`node_free:BOOLEAN,`
`t_finished:BOOLEAN,`	`m_set:BOOLEAN,`	`node_task:TASKS #];`
`t_start:NATURAL,`	`m_slot:NATURAL #];`	
`t_clock:NATURAL,`		
`t_comp:NATURAL,`		
`t_node:NODES #];`		

Fig. 2. Declaration of task, message and node records

The defined task record, for instance, stores the parameter information of a single task t_i. The variable `t_started` indicates whether t_i has already started or not. Because a tasks' computation time can last for more than one time step, the variable `t_finished` indicates if task t_i has finished or not. In combination with the variable `t_clock`, which holds the actual task computation duration, the current status of a task can be exactly described. To obtain a time schedule, each task needs to store its calculated starting time. This is done by the variable `t_start`. Furthermore, `t_comp` stores the given computation duration of task t_i, and `t_node` represents the node a task is allocated to. We specify and store the parameters of all tasks defined for the given precedence graph $\mathcal{G}$ as an array of records, `TASKARRAY : TYPE = ARRAY TASKS OF taskcalendar;`.

Modeling precedence relations requires a distinction between task and message. For each given task and message, the direct predecessor is stored. A predecessor of a task is defined as a message, except if the task is the starting task (source of precedence graph). The source task has no predecessors. For instance, a task t_i may receive two messages. These messages are stored as predecessors of that task. A predecessor of a message is always a single task. These precedence relations are represented via an array structure, as `PREC_TASK : ARRAY INDEX OF MESSAGES` and `PREC_MSG : ARRAY INDEX OF TASKS`.

The allocation of task to nodes, $\tau : T \rightarrow 2^M$ (cf. section 2), is specified by an array structure as `NODETASKS: TYPE = ARRAY INDEX OF NODES`, while the unique precedence relations of each single task, is modeled as `prec_TASK1 : PREC_TASK = [[i:INDEX]Edge0]`.

The precedence relations for all messages as well as the allocation of tasks to nodes is modeled in the same way, `prec_Edge1: PREC_MSG = [[i:INDEX]Task0]`, and `Task1_node: NODETASKS = [[i:INDEX]Node1]`, respectively.

Furthermore, two functions `getPrecTask` and `getPrecMsg` are needed by the transitions as a condition request about a tasks' or a message's predecessors. The `getPrecTask` function, for instance, is used by a certain message to check whether its predecessor (namely the task who sends this message) is already finished (`t_finished = TRUE`). Hence, `getPrecTask` has two parameters: the message `m`, whose predecessors' condition is requested and the task record array `taskarry`. If, and only if the predecessor task (or sender task) is already finished the functions returns `TRUE`. This scenario is specified in SAL as follows:

```
getPrecTask(m : MESSAGES, taskarray : TASKARRAY) : BOOLEAN =
IF(m=Edge0 AND(EXISTS(j:INDEX1):taskarray[prec_Edge0[j]].
        t_finished=FALSE)) THEN FALSE
ELSIF (m=Edge1 AND(EXISTS(j:INDEX1):taskarray[prec_Edge1[j]].
        t_finished=FALSE)) THEN FALSE
```

The function `getPrecMsg(t:TASKS, msgarray:MSGARRAY) : BOOLEAN` detects for a certain task t_i if all predecessors (namely all messages m_j with $\mathcal{D}(t_x, m_j) = t_i$) are already set. If all predecessors messages are already scheduled, the function returns `TRUE`. Initially, only the source tasks fulfills this request. Thus, it can be said, that the schedule is build according to the precedence graph from source to sink. A further function `getNodeTask(t:TASKS):NODES` is modeled to detect and return the node n, a certain task t_i is allocated to.

3.2 Basic Module

The basic construct in SAL is a module. A module contains the definition of variables (including, local, global, input and output), the initial state, and the transition relations. The discrete-time variable $Time$ is used for specifying the global system time. Thus, the state space is finite.

```
scheduler : MODULE = BEGIN
 GLOBAL currenttaskarray : TASKARRAY; currentmessagearray : MSGARRAY;
        currentnodearray : NODEARRAY; Time : TIME; Bus_free:BOOLEAN
```

For instance, the initial state for the simple precedence graph from 2, is specified in SAL as follows:

```
INITIATLIZATION
  currenttaskarray[Task0].t_comp = 2.0;
  currenttaskarray[Task0].t_node = Node0; ...
  Time = 0; Bus_free = TRUE; ...
  (FORALL (i:TASKS): currenttaskarray[i].t_started = FALSE);
  (FORALL (i:TASKS): currenttaskarray[i].t_finished = FALSE);
```

```
(FORALL (i:TASKS): currenttaskarray[i].t_start = 0);
(FORALL (i:TASKS): currenttaskarray[i].t_clock = 0);
(FORALL (i:MESSAGES): currentmessagearray[i].m_started = FALSE);
(FORALL (i:MESSAGES): currentmessagearray[i].m_set = FALSE);
(FORALL (i:MESSAGES): currentmessagearray[i].m_slot = 0);
(FORALL (i:NODES): currentnodearray[i].node_free = TRUE);
(FORALL (i:NODES): currentnodearray[i].node_task = Task3);
```

Initially, the computation time is set for each task to `t_comp=2.0`. Tasks are allocated to nodes `currenttaskarray[Task0].t_node=Node0`. The global variable `Time` is initially set to 0. The time-triggered communication bus is represented by the variable `Bus_free`, which indicates whether, at a certain point in time, the bus, respectively the current time slot, is allocated by a node or not. Initially, the bus is not used. Initialization of the record arrays for tasks, messages and nodes can be described as follows: Initially, none of the tasks has started, thus the variable `t_started` is set to `FALSE`. No task has finished (`t_finished = FALSE`) and starting time variable (`t_start`) is set to 0. The clock variable `t_clock` is also set to 0. The initialization for all given messages is done in the same way. All messages are not started (`m_started = FALSE`). The variable `m_set = FALSE` points out that no messages has been scheduled yet, which implies that also no slot has been allocated by a certain message (`m_slot = 0`). For the node record the parameter `node_free` indicates that the current nodes' CPU is not allocated by a task. The parameter `node_free` is initially set to the sink task in the given precedence graph $\mathcal{G}$.

3.3 Transitions

State transitions are specified in SAL via guarded commands. Here, the [character introduces a set of guarded commands, which are separated by the [ ] symbol. A SAL guarded command is eligible for execution in the current state if its guard (i.e., the part before the `-->` arrow) is true. The SAL model checker nondeterministically selects one of the enabled commands for execution at each step. In case no command is eligible, the system is deadlocked. State variables are unprimed before execution of a command and primed in the new state, that is after the execution command.

The scheduling algorithm is encoded in SAL using six different transition relations. Four transition relations (`start_task`, `run_task`, `change_task`, `end_task`) are used to model task scheduling, while `start_message`, and `end_message` encodes message scheduling on bus level.

3.3.1 Start Task Transition

The Start Task Transition allows for starting all tasks, which comply to the following conditions: (1) the CPU is not yet allocated by another task on the same node, (2) the task has either finished or started and (3) all predecessors (input messages) are already set (arranged by the function `getPrecMsg`).

```
[([](i:TASKS): start_transition:
currentnodearray[getNodeTask(i)].node_free = TRUE AND
currenttaskarray[i].t_started = FALSE AND
getPrecMsg(i,currentmessagearray) = TRUE    -->
currenttaskarray'= currenttaskarray WITH [i].t_started:=TRUE
                   WITH [i].t_clock:=0 WITH [i].t_start:=Time;
currentnodearray'= currentnodearray
                   WITH [getNodeTask(i)].node_free:=FALSE
                   WITH [getNodeTask(i)].node_task:=i)
```

Any task t_i, whose corresponding node is free (first line of code) and that has not started yet (second line), is allowed to be started (`currenttaskarray[i].t_started = TRUE`). The clock is set to 0 and the starting time is set to the actual time (`currenttaskarray[i].t_start = Time`). Furthermore, the node's resource is allocated by the current task t_i, therefore, `node_free` variable is set to FALSE.

3.3.2 Run Task Transition

The Run Task Transition allows tasks to run on their allocated node (nodes' CPU) by incrementing the time variable `Time`, respectively the task' clock variable `clock` by one. Basically, the run transition controls the progress of time. Each task, which has already started and whose computation duration is not actually reached, can progress its execution time, controlled by the `t_clock` variable. The first part of the pre-condition (i.e., first disjunct) states that the current task t_i has started yet. The value of task clock variable `t_clock` has to be less than the value of the computation time variable `t_comp`, otherwise the task would be a candidate for the end transition. Three additional conditions, for excluding the possibility of executing other transitions, have to be satisfied for enabling the run transition: (1) No other task is able to start on another node (handled by the `start_transition`), (2) No other task is able to be finished (handled by the `end_transition`) and (3) No message is able to start (handled by the `start_message_transition`). The second part of the disjunction in the precondition specifies the scenario when the current running task is interrupted by another (cf. 3.3.3). These conditions states that (1) another task can be started on the same node, (2) no other task can be finished, (3) no message is able to start.

These pre-conditions describe the exact situation in which the `run_transition` is used, that is, the situation in which tasks executes and time is allowed to pass. Due to the lack of space the extensive code is not described in detail here.

```
run_transition:
(Compare textual description for preconditions above) -->
currenttaskarray'[Task0].t_clock = IF currenttaskarray[Task0]
                                                 .t_started = TRUE AND
      currenttaskarray[Task0].t_clock < currenttaskarray[Task0].t_comp
    THEN currenttaskarray[Task0].t_clock+1
    ELSE currenttaskarray[Task0].t_clock ENDIF;
Time'=Time+1;
```

3.3.3 Change Task Transition

The Change Task Transition allows for stopping an already started task t_j by another task t_i, which is allocated to the same resource n_i. This may be necessary if the precedence graph $\mathcal{G}$ consists of concurrent task precedence relations, which might originate two competitively tasks (share the same resource) at the same point in time. In order to find an optimal solution, in terms of total length of the final schedule, it might be better to interrupt such an already started task t_j by another one. The guarded command reflects these requirements by determining a potential starting task `currentaskarray[i]` whose resource is currently allocated (`currentnodearray[getNodeTask(i)].node_free = FALSE`).

```
([](i:TASKS):   change_transition:
currentnodearray[getNodeTask(i)].node_free = FALSE AND
currenttaskarray[i].t_started = FALSE AND
getPrecMsg(i,currentmessagearray) = TRUE    -->
currenttaskarray'= currenttaskarray WITH [i].t_started:= TRUE
  WITH [i].t_clock:=0 WITH [i].t_start:=Time
  WITH [currentnodearray[getNodeTask(i)].node_task].t_started:=FALSE
  WITH [currentnodearray[getNodeTask(i)].node_task].t_clock:=0
  WITH [currentnodearray[getNodeTask(i)].node_task].t_start:=0;
currentnodearray'= currentnodearray WITH [getNodeTask(i)].node_free:=F
   WITH [getNodeTask(i)].node_task:=i)
```

If the preconditions are satisfied, the task, currently using the required resource (`currenttaskarray[currentnodearray[getNodeTask(i)]]`), is interrupted, and its starting time (`...t_start:=0`) and computation counter (`...t_clock:=0`) is reseted. This steams from the fact that we do not consider here any preemption scheduling possibilities. The concurrent task (`currenttaskarray'[i]`) is started instead and allocated to the shared resource (`currentnodearray'=...`).

3.3.4 End Task Transition

The End Task Transition ensures that a task t_i can be finished and the resource released. The guarded command detects whether the task execution time (represented by `currenttaskarray[i].t_clock`) equals the tasks' computation time (`currenttaskarray[i].t_comp`).

```
([](i:TASKS): endtransition:
currenttaskarray[i].t_started = TRUE AND
currenttaskarray[i].t_clock = currenttaskarray[i].t_comp AND
currenttaskarray[i].t_finished = FALSE -->
  currenttaskarray'[i].t_finished = TRUE;
  currentnodearray'= (currentnodearray
            WITH [getNodeTask(i)].node_free:=TRUE)
            WITH [getNodeTask(i)].node_task:=Task3)
```

In case these preconditions are fulfilled the tasks' execution is finished and the nodes' resource is released for the possible execution of the next task.

3.3.5 Start Message Transition

The Start Message Transition enables sending of a message on the bus in the next available time slot of the underlaying time-triggered protocol. As a precondition the next possible slot needs to be free (not already allocated) by another message. This is checked by the variable `Bus_free =TRUE`. Furthermore, the message should not be started and not finished yet, (`currentmessagearry[i].m_started=FALSE, currentmessagearry[i]. m_set = FALSE`). However, the task, which sends the message, needs to be finished. The `start_message_transition`, like the `run_task_transition`, is able to progress time by one time unit. Thus, it needs to be checked, whether other transitions are able to be handled first. Therefore, two more conditions, introduced by `NOT (EXISTS...)`, cover these constrains: (1) No task is able to start (handled by the `start_task_transition`) and (2) No task is able to be finished (handled by the `end_task_transition`).

```
([](i:MESSAGES): start_message_transition:
 currentmessagearray[i].m_set = FALSE AND
 currentmessagearray[i].m_started = FALSE AND
 Bus_free = TRUE AND
 getPrecTask(i,currenttaskarray)=TRUE AND
 (NOT(EXISTS(j:TASKS):currentnodearray[getNodeTask(j)].node_free=TRUE AND
       currenttaskarray[j].t_started = FALSE AND
       getPrecMsg(j,currentmessagearray)=TRUE)) AND
(NOT(EXISTS(k:TASKS):currenttaskarray[k].t_started = TRUE AND
   currenttaskarray[k].t_clock = currenttaskarray[k].t_comp AND
   currenttaskarray[k].t_finished = FALSE)) -->
currentmessagearray'=currentmessagearray WITH [i].m_started:=TRUE
                                         WITH [i].m_slot:=Time;
Bus_free' = FALSE;
Time'=IF(NOT (EXISTS(j:TASKS):currenttaskarray[j].t_started=TRUE AND
              currenttaskarray[j].t_finished=FALSE))
    THEN  Time+1 ELSE Time  ENDIF;)
```

If the preconditions are satisfied the current message, (`currenttaskarray[j]`), is started and allocated to a certain slot. Hence, the bus is blocked and if there are no further tasks which need to be started or finished the time is incremented.

3.3.6 End Message Transition

The End Message Transition ensures that a message m_i can be finished and the bus resource is released. The guarded command represents this requirement and checks, if time has elapsed since starting the message / slot allocation (cf. `Time = currentmessagearray[i].m_slot+1`). If these conditions are satisfied the bus resource is released and the message is marked as set (`currentmessagarray'[i].m_set = TRUE`).

```
([](i:MESSAGES): endmessagetransition:
currentmessagearray[i].m_set = FALSE AND
currentmessagearray[i].m_started = TRUE AND
Time = currentmessagearray[i].m_slot+1  -->
currentmessagearray'[i].m_set = TRUE;   Bus_free' = TRUE)
```

With the combination of all transitions it is possible to generate an integrated task and message schedule which is optimized for minimizing the maximum latency. As specified in the `run_task_transition`, for example, the order of transition execution is controlled. This is done due to the optimization criteria as well as for complying with all kinds of precedence graph characteristics (e.g., sequential or concurrent precedence graphs).

4 Analysis and Results

In this section we present results from our experiments using the SAL's model checkers for the development of an integrated task and message schedule for hard-real time systems. Our experiments were performed on an Intel(R) Pentium(R) 4CPU 2.80GHz and 2,49GB RAM.

The primary function of a model checker is the analysis of a specified model with respect to a given property. The model checker returns either *verified or* falsified, depending whether a given property is fulfilled by the model or not. In the latter case, the model checkers usually output a counterexample. We use this capability to compute a combined task and message schedule that is optimal in terms of minimizing the maximum latency.

By failing the property we obtain a counterexample containing a task and message schedule. This schedule might not be optimal, concerning end to end latency, because the transition system is allowed to take transitions at which no time is consumed. To obtain an optimal task and message schedule under consideration, we further use a binary search for finding that solution. To find such a counterexample we can either use SAL's bounded model checker [11], sal-bmc, or one of SAL's symbolic model checkers, sal-smc or, sal-wmc [12]. We want a schedule that guarantees that all tasks are finished and all messages are sent, that is we look for a witness for the CTL formula $EF(\forall i : TASKS.\ currenttaskarray[i].t\_finished = TRUE)$. The condition that all message are sent is implicitly contained in this formula, since all tasks finished implies that all messages have been sent. By model checking the negation of above formula, we obtain a counterexample that contains the solution to the task-message scheduling problem. This negation of the above CTL formula is specified in SAL as the following theorem, stating that in (reacheable) every state there are some tasks that have not finished. A counterexample to this formula is generated if a state is reached in which all tasks are finished. The value of the state variables in the last step of the counterexample gives us the desired schedule fulfilling the requirement of minimizing the maximum latency.

```
th: theorem schedule |- AG(EXISTS (i:TASKS):currenttaskarray[i]
                                            .t_finished=FALSE);
```

For analysis and demonstration of operation, we use the simple example explained in section 2. We invoke SAL's symbolic model checker with the command `sal-smc -v 3 scheduler th`. SAL outputs a counterexample, which is explained in the following. Some interesting properties are listed below. The verification time of `sal-smc` while finding a counterexample is measured with $1,109$ seconds. The Model Checker explores 93952 states with a total execution time of $7,845$ seconds.

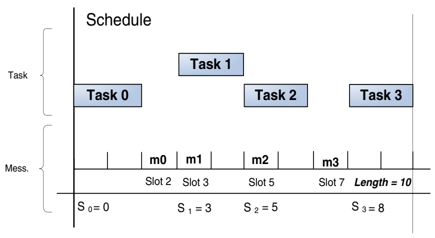
The symbol model checker shows that there is a state that can be reached within 24 steps, in which all tasks are finished. This complies to the shortest counterexample possible. Because of the given functions `getPrecTask` and `getPrecMsg`, as described in section 3.1, the schedule is modeled - in the words of a precedence graph - from source to sink. In case of any resource constraints, either bus or CPU restrictions, the model checker performs different possible schedules. For example, there are two tasks, task t_1 and task t_2 allocated to the same resource which are able to be started. Thus, three different possibilities are given. One possibility allocates primary task t_1 to the shared resource, in the other one the task t_2 is preferred. The third possible solution would be if none of them would allocate this resource. However, the third solution would not be feasible, because of the optimality requirement of minimizing the maximum latency. The `start_task_transition` enables these possibilities by defining these possibilities as next states s'_1, s'_2 and s'_3. Additionally the `change_task_transition` handles the further possibility of stopping and replacing an already started task. This is useful to minimize the overall scheduling length.

Thus, by reaching a state, that fails the correctness property (complies to the situation in which all tasks are finished), it would implicitly indicate that there are no counterexamples with less than the certain number of steps. Hence, there are no shorter schedules reachable. As shown in the counterexample for our example, 24 transitions steps are necessary to find a schedule, in which all tasks have been scheduled optimally.

```
Step 24: --- System Variables (assignments) ---
currenttaskarray[Task0].t_clock = 2
currenttaskarray[Task0].t_comp = 2
currenttaskarray[Task0].t_finished = true
currenttaskarray[Task0].t_node = Node0
currenttaskarray[Task0].t_start = 0
currenttaskarray[Task0].t_started = true...
currentmessagearray[Edge0].m_set = true
currentmessagearray[Edge0].m_slot = 2
currentmessagearray[Edge0].m_started = true
currentmessagearray[Edge1].m_set = true ...
Time = 10
Bus_free = true
```

The counterexample comprises the different global variables, calculated by the model checker. From this counterexample a schedule $\gamma = \{t_i \mapsto \gamma_i | \forall t_i \in T\}$ for our simple example (compare section 2) is extracted as: $\gamma = \{t_0 \mapsto \langle 0, \{sl_2, sl_3\}\rangle, t_1 \mapsto \langle 3, \{sl_5\}\rangle, t_2 \mapsto \langle 5, \{sl_7\}\rangle, t_3 \mapsto \langle 8, \{\}\rangle\}$

This schedule is illustrated in figure 3a. As explained in section 2, the tasks $Task1$ and $Task2$ are allocated to the same node: $Node1$. Hence, the execution ordering needs to be sequential for that node (compare figure 3a). The optimal schedule length for the given tasks is 10 time units. This is given by the global variable `Time` in the above counterexample. The `Time` variable is just influenced by the two transitions `run_task_transition` and `start_message_transition`. As explained in section 3.3, these transitions guarantee that the time only proceeds, if no other transitions are able to be handled. That is why the current time reached in any certain state, represented by the variable `Time` equals the schedule length.



(a) Calculated Task and Message Schedule

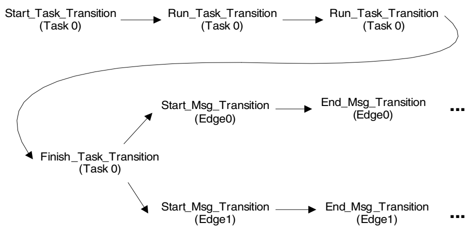

Start_Task_Transition (Task 0)
Run_Task_Transition (Task 0)
Run_Task_Transition (Task 0)
Finish_Task_Transition (Task 0)
Start_Msg_Transition (Edge0)
End_Msg_Transition (Edge0)
Start_Msg_Transition (Edge1)
End_Msg_Transition (Edge1)
...
...
(b) Ordering of transitions

Fig. 3. Final Schedule and Ordering of transitions

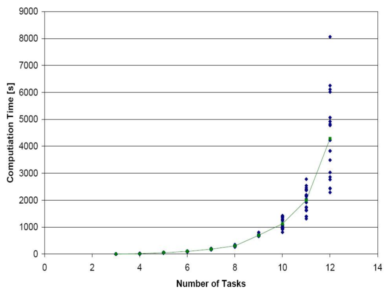
Although the transitions are written in arbitrary order, they cannot be executed non-deterministically due to undesirable effects on the integrated schedule. Thus, an ordering has to be chosen in such a way to reflect the constraints posed on task execution and message transmission as given by the precedence graph. For the example in section 2 the possible execution ordering of transitions is depicted in figure 3b. The given precedence graph $\mathcal{G}$ is traversed from source to sink. The source task(s) t_{source} are characterized by the non-existence of any predecessors (input messages). Thus, the start task transitions for such source tasks are executed first. Whenever the given precedence graph $\mathcal{G}$ allows for different possible solutions, characterized by the concurrency to a shared resource (either the time-triggered bus or a nodes' CPU), we use the model checker's capabilities (state space exploration) to explore all interleaved possibilities. The ordering of transitions is caused by this requirement. Figure 3.3 points out that both Edge0 (message 0) and Edge1 (message1) may access the next available time slot. Thus, both ways are handled via different next states s'. Generally, from each given state s the transitions generate a next state s' with all necessary scheduling combinations which are possible, according to the given precedence graph $\mathcal{G}$.

4.1 Results of Experiment Series

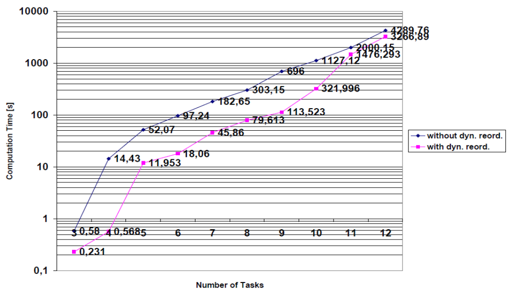
To get a better understanding of complexity in terms of precedence graph structure and verification time, a graph generator tool was written in Java. The verification time can be seen as the CPU time for verifying an integrated task and messages schedule using SAL 3.0. These runtimes are given in seconds and were measured for 20 randomly generated graphs consisting out of N number of tasks ranging from 3 to 12. Each graph may have multiple starting nodes and multiple ending nodes. The complete structure is calculated randomly. The proportion of task and message is 1:1.

All these numbers, given in figure 4a, must be taken with caution. They reflect the performance of `sal-smc` on the specific formalization of the integrated task and message scheduling problem specified in SAL. But, never the less, the expected trend can be confirmed. Each verification time displayed in the table is the average value of all 20 graphs consisting of N tasks. The standard deviation reflects the mean variation of values from the average value of all 20 graphs. As it can be seen by the verification time, increasing the number of task in the randomized precedence graphs would increase the verification time exponentially. Besides, the standard deviation is increased dramatically as well. While having a standard deviation of $9,05$s for graphs with 6 tasks, the mean variation from the average value for precedence graphs with 12 tasks was calculated with $1619,60$s.

In the following the computation time is shown by a logarithmically diagram. The verification times comply the average value of all 20 graphs. It can be seen, that starting from precedence graphs containing a minimum of 6 tasks without dynamic reordering, a linear correlation of verification time can be identified. Thus, for 13 tasks an average verification time of around 10000 s can be expected (cf. figure 4b).



(a) Computation Time of Experiment Series



(b) Comparison with dynamic reordering

Fig. 4. Results of Experiment Series

Model Checking has been proven to be a powerful tool in calculation of integrated task and message scheduling. However, efficient model checking of problems with huge state spaces is only possible with efficient representation of the model itself. Ordered Binary Decision Diagrams (OBDDs) allow an efficient symbolic representation of the model. As the size of the OBDDs and also the computation time depends on the order of the input variables, dynamic reordering strategies may accelerate the process of computation and increases the efficiency of computation. Figure 4b illustrates the positive effect of dynamic reordering on the computation time.

5 Conclusion and Future Work

We have presented an automatic approach to combined task and message scheduling for TDM-based applications that allows to compute an integrated task and message schedule. We described the modeling concepts for abstracting the given problem symbolically in SAL, together with the generation of an optimal schedule under the given requirement of minimizing the maximum latency.

Experimental results have demonstrated how the latest generation of model-checking tools meets the challenges of providing both a convenient modeling language and the performance to solve given scheduling problems. However, it remains to evaluate whether our approach scales up to large industrial applications. We are currently implementing a heuristic approach for state-space reduction to scale up for larger application. Furthermore weighted state-space reduction improvements are considered.

References

1. Pop, P., Eles, P., Peng, Z.: Scheduling with optimized communication for time-triggered embedded systems. In: CODES 1999, pp. 178–182. ACM Press, New York (1999)
2. Tovar, E., Vasques, F.: From task scheduling in single processor environments to message scheduling in a profibus. In: IPPS/SPDP Workshops, pp. 339–252 (1999)
3. Voss, S., Sorea, M., Echtle, K.: Symbolic task and message scheduling for time-triggered networks (in preparation, 2008)
4. Farcas, E., Farcas, C., Pree, W., Templ, J.: Transparent distribution of real-time components based on logical execution time. SIGPLAN Not. 40(7), 31–39 (2005)
5. Tindell, K., Clark, J.: Holistic schedulability analysis for distributed hard real-time systems. Microprocessing and Microprogramming - Euromicro Journal (Special Issue on Parallel Embedded Real-Time Systems) 40, 117–134 (1994)
6. Abdelzaher, T.F., Shin, K.G.: Combined task and message scheduling in distributed real-time systems. IEEE Trans. Parallel Distrib. Syst. 10(11) (1999)
7. Jensen, A.R., Lauritzen, L.B., Laursen, O.: Optimal task graph scheduling with binary decision diagrams (2004)
8. Metzner, A., Fränzle, M., Herde, C., Stierand, I.: Scheduling distributed real-time systems by satisfiability checking. In: RTCSA 2005, Washington, DC, USA, pp. 409–415. IEEE Computer Society Press, Los Alamitos (2005)
9. de Moura, L., Owre, S., Rueß, H., Rushby, J., Shankar, N., Sorea, M., Tiwari, A.: Tool presentation: SAL2. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114. Springer, Heidelberg (2004)
10. Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
11. de Moura, L., Rueß, H., Sorea, M.: Bounded model checking and induction: From refutation to verification. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 14–26. Springer, Heidelberg (2003)
12. Shankar, N., Sorea, M.: Counterexample-driven model checking. Technical Report SRI-CSL-03-04, SRI International (2003)

Incremental Reasoning for Multiple Inheritance[*]

Johan Dovland, Einar Broch Johnsen, Olaf Owe, and Martin Steffen

Department of Informatics, University of Oslo, Norway
{johand,einarj,olaf,msteffen}@ifi.uio.no

Abstract. Object-orientation supports code reuse and incremental programming. Multiple inheritance increases the power of code reuse, but complicates the binding of method calls and thereby program analysis. Behavioral subtyping allows program analysis under an *open world assumption*; i.e., under the assumption that class hierarchies are extensible. However, method redefinition is severely restricted by behavioral subtyping, and multiple inheritance often leads to conflicting restrictions from independently designed superclasses. This paper presents an approach to incremental reasoning for multiple inheritance under an open world assumption. The approach, based on a notion of *lazy behavioral subtyping*, is less restrictive than behavioral subtyping and fits well with multiple inheritance, as it incrementally imposes context-dependent behavioral constraints on new subclasses. We formalize the approach as a calculus, for which we show soundness.

1 Introduction

Object-orientation supports code reuse and incremental programming through inheritance. Class hierarchies are extended over time as subclasses are developed and added. A class may reuse code from its superclasses but it may also specialize and adapt this code by providing additional method definitions, possibly overriding definitions in superclasses. This way, the class hierarchy allows programs to be represented in a compact and succinct way, significantly reducing the need for code duplication. *Late binding* is the underlying mechanism for this incremental programming style; the binding of a method call at run-time depends on the actual class of the called object. Consequently, the code to be executed depends on information which is not statically available. Although late binding is an important feature of object-oriented programming, this loss of control severely complicates reasoning about object-oriented programs.

Behavioral subtyping is the most prominent solution to regain static control of late-bound method calls (see, e.g., [21, 1, 20]), with an *open world assumption*; i.e., where class hierarchies are extensible. This approach achieves incremental reasoning in the sense that a subclass may be analyzed in the context of previously defined classes, such that previously proved properties are ensured by additional verification conditions. However, the approach restricts how methods may be redefined in subclasses. To avoid reverification, any method redefinition must *preserve* certain properties of the method which is redefined. In particular, this applies to the method's contract; i.e., the pre- and

[*] This research is partially funded by the EU project IST-33826 CREDO: Modeling and analysis of evolutionary structures for distributed services (http://credo.cwi.nl).

postcondition for its body. This contract can be seen as a description of the promised behavior of all implementations of the method. Unfortunately, this restriction hinders code reuse and is often violated in practice [30]; for example, it is not respected by the standard Java library definitions.

Multiple inheritance offers greater flexibility than single inheritance, as several class hierarchies can be combined in a subclass. However, it also complicates language design and is often explained in terms of complex run-time data structures such as virtual pointer tables [31], which are hard to understand. Formal treatments are scarce (e.g., [29,8,5,14,32]), but help clarify intricacies, thus facilitating design and reasoning for programs using multiple inheritance. Multiple inheritance also complicates behavioral reasoning, as name conflicts may occur between methods independently defined in different branches of the class hierarchy.

Work on behavioral reasoning about object-oriented programs has mostly focused on languages with single inheritance (see, e.g., [27,28,7]). It is an open problem how to design an incremental proof system for multiple inheritance under an open world assumption, without severely restricting code reuse. In this paper we propose a solution to this problem. The approach extends *lazy behavioral subtyping*, which was originally developed for single inheritance systems [13] to allow more flexible code reuse than reasoning systems based on behavioral subtyping. Our approach applies to a wide class of object-oriented systems, relying on the assumption of a *healthy* binding strategy, which is needed for incremental reasoning. Healthiness may easily be imposed on non-healthy binding strategies. The approach is formalized as a syntax-driven inference system, for which we show soundness, combines deductive style program logic with incremental program development, and is well-suited for program development environments. Proofs and a more detailed example may be found in [12].

Paper overview. Sect. 2 introduces late binding and multiple inheritance, Sect. 3 proof environments for behavioral reasoning, and Sect. 4 presents the inference system for incremental reasoning. Sect. 5 discusses methodological aspects, Sect. 6 discusses related work, and Sect. 7 concludes the paper.

2 Late Binding and Multiple Inheritance

An object-oriented kernel language is given in Fig. 1, based on Featherweight Java [16]. For simplicity, we let expressions e be without side-effects and assume that fields f have (locally) distinct names, methods with the same name have the same signature (i.e., no method overloading), class names are unique, programs are well-typed, there is read-only access to the formal parameters of methods, as well as **this**, and we ignore the

$$\begin{array}{ll}
P ::= \overline{L}\ \{t\} & L ::= \mathbf{class}\ C\ \mathbf{extends}\ \overline{C}\ \{\overline{f}\ \overline{M}\} \\
M ::= m\ (\overline{x})\{t\} & e ::= \mathbf{new}\ C \mid b \mid v \mid \mathbf{this} \mid e.m(\overline{e}) \mid m(\overline{e}) \mid m(\overline{e})@C \\
v\ ::= f \mid f@C & t\ ::= v := e \mid \mathbf{return}\ e \mid \mathbf{skip} \mid \mathbf{if}\ b\ \mathbf{then}\ t\ \mathbf{else}\ t\ \mathbf{fi} \mid t;t
\end{array}$$

Fig. 1. The language syntax, with class names C and method names m. Expressions e include fields, **this**, object creation, Boolean expressions b, and method calls. Whitespace is used for list concatenation (i.e., $\overline{e}$ is a list and $e\ \overline{e}$ a non-empty list of expressions).

types of fields and methods. Two notable differences to Featherweight Java are multiple inheritance and a corresponding form of static method calls. These are explained below. (For brevity, we do not explain standard language features in detail.)

A class C extends a list $\overline{C}$ of superclass names with fields $\overline{f}$ and methods $\overline{M}$, where $\overline{C}$ consists of unique names. We say that C *defines* a method m if the set $\overline{M}$ contains an implementation of m. Let a partial function $body(C,m)$ return this implementation (so $body(C,m)$ is undefined if m is not in $\overline{M}$). For any superclass B of C and method m defined in B, we say that C *inherits* m from B if C does not define m, otherwise m is *overridden* in C. We say that a class C_1 is *below* class C_2 (written $C_1 \leq C_2$) if C_1 and C_2 are the same class or if C_1 extends a class below C_2. Furthermore, C_2 is *above* C_1 if C_1 is below C_2. A *subclass* is below a *superclass*. Two classes are *related* if one is below the other.

There are two kinds of method calls. A *static call* $m(\overline{e})@C$ may occur in a class below C, and it is bound above C at compile time. This statement generalizes the call to the superclass found in languages with single inheritance. In a *remote call* $e.m(\overline{e})$, the object e receives a call to m with actual parameters $\overline{e}$. For convenience, we write $e.m(\overline{e})$ or simply $e.m$ instead of $v := e.m(\overline{e})$ if the result is not needed. Explicit self-calls, written $m(\overline{e})$, are late-bound to **`this`**. Similarly, $f@C$ binds a field f above C.

2.1 Name Conflicts and Healthiness

Inheritance relates classes in a class hierarchy. For single inheritance this hierarchy forms a tree, whereas for multiple inheritance, the hierarchy forms a directed, acyclic graph. In the single inheritance tree, *vertical* name conflicts occur when a subclass overrides a method from a superclass. The *binding strategy* for method calls must resolve such conflicts. Late binding or dynamic dispatch selects the method body to be executed at run-time, depending on the callee's run-time class: the selected body is found by the *first matching definition* of the method above the actual class of the object. In multiple inheritance class hierarchies there are also *horizontal* name conflicts. These occur when different definitions of the same method are found above a given class, depending on the chosen path through the hierarchy. More elaborate binding strategies are needed to resolve horizontal conflicts. Some binding strategies are infeasible, as they contradict incremental program development. This is illustrated by the following example.

Example 1. We consider a class hierarchy for a bank account system, given in Fig. 2. Potential problems with horizontal name conflicts are illustrated by the classes in Fig. 3, sketching an implementation of the account system. (A more detailed implementation

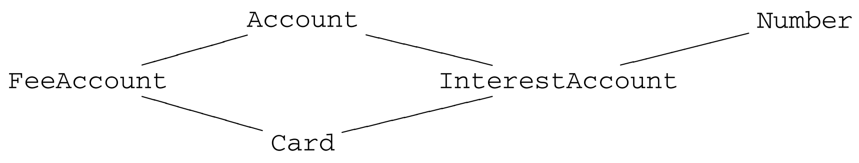


Fig. 2. A multiple inheritance class hierarchy for an account system. The inheritance relation is indicated by lines, e.g., class `FeeAccount` inherits from `Account`.

```
class Account { int bal = 0;               class Number { int num;
  deposit(int x) {...;update(x)}             update(int x) {num = x }
  withdraw(int x) {...;update(-x)}           increase(int x) {update(num+x)}}
  update(int y) {...;bal=bal+y;...}}

class InterestAccount extends Account Number { int fee;
 addInterest(int x y) {...; deposit(x);increase(y)}
 withdraw(int x) {withdraw(x)@Account;if bal<0 then update(-fee) fi}}

class FeeAccount extends Account { int fee;
 withdraw(int x) {withdraw(x)@Account;update(-fee) }
 update(int y) {...; bal=bal+y;...}}

class Card extends FeeAccount InterestAccount {
 withdraw(int x) {withdraw(x)@InterestAccount;update(-fee@FeeAccount)}}
```

Fig. 3. Implementation sketches of the classes in Fig. 2

is available in the extended version of this paper [12].) Class `Account` implements basic facilities for depositing and withdrawing money. The actual manipulation of the balance is implemented by a method `update`. Class `Number`, developed independently of `Account`, provides functionality for manipulating a number `num`. These two classes are inherited by the subclass `InterestAccount`, where field `num` plays the role of the current interest rate. Method `addInterest` increases the interest rate after depositing a value. For `InterestAccount`, inheritance of `Account` and `Number` gives a horizontal name conflict for method `update`. The behavior of the two versions of `update` is completely different, which means that the behavior specified by `increase` in `Number` will not hold in the subclass, if the self-call to `update` in `increase` is bound to `Account`. Thus, in order to support incremental design, the self-call in `Number` should bind to the definition in `Number`, and correspondingly for the self-call in `Account`should bind to the definition in `Account`.

One solution to resolve horizontal name conflicts is *explicit resolution* specified as part of an inheritance list; e.g., to use qualification or renaming as in C++ [31], Eiffel [23], and POOL [2]. However, it might be undesirable to force the programmer to modify method names, making programs more difficult to understand and maintain. We generalize this approach and decorate each self-call with a *binding clause* restricting the binding space. Such a clause may represent a specific name resolution strategy, or be explicitly provided by the programmer. This way, the approach of this paper is applicable to several resolution strategies. Binding clauses allow us to consider horizontal name conflicts as a natural feature of multiple inheritance. In particular when using libraries, the programmer cannot be expected to know (or resolve) potential name conflicts of, e.g., auxiliary methods in the libraries. To support incremental program development and reasoning, we impose the following *healthiness condition* on the binding strategy:

– a self-call made by a method defined in C must bind to a class related to C, and
– a remote call $x.m$, where x has C as declared class, must bind to a class related to C.

It is easy to see that healthiness removes accidental overriding of methods, due to unfortunate binding. Let $C\#m$ denote a call to m where the binding is restricted to classes related to C. In Example 1, if the call to `update` in `Number` is replaced by `Number#update`, the call becomes healthy. When executed in an instance of `InterestAccount`, the call will bind related to `Number`. For the rest of the paper, we use the convention that a self-call to m made by a method defined in C is understood as $C\#m$. Similarly, a remote call $x.m$ with C as the declared class of x, is understood as $x.C\#m$. As static calls are inherently healthy, this ensures healthy binding. A particular binding strategy is given below. We here assume that the notation $C\#m$ and $x.C\#m$ is introduced during static analysis, but it could also be made available to the programmer.

2.2 The Binding of Method Calls and Fields

For the reasoning system, we need an explicit definition of a healthy resolution strategy. In this paper, we formalize the strategy by a function *bind* defined below. Other definitions of *bind* are possible and would lead to variations of the calculus. A call to a method m is bound with respect to a *search class* D; i.e., $bind(D,m)$, where the search for a definition of m starts in D. Following [9, 11, 17], ambiguities are solved by fixing the order in which inherited classes are searched, e.g., from left to right. Let *Cid* and *Mid* denote class and method names. To make the representation of class hierarchies compact, a class name is bound to a tuple $\langle \overline{C}, \overline{f}, \overline{M} \rangle$ of type *Class*, where $\overline{C}$, $\overline{f}$, and $\overline{M}$ are accessible by the observer functions *inh*, *fields*, and *mtds*, respectively. This binding strategy can be defined by a partial function *bind*: $List[Cid] \times Mid \to Cid$:

$$\begin{array}{lll} bind(nil, m) & \triangleq \bot & \\ bind(D\,\overline{D}, m) & \triangleq D & \text{if } m \in D.mtds \\ bind(D\,\overline{D}, m) & \triangleq bind(D.inh\,\overline{D}, m) & \text{otherwise,} \end{array}$$

where *nil* denotes the empty list and $D.inh\,\overline{D}$ reduces to $\overline{D}$ when $D.inh$ is empty. This strategy is not healthy, since a self-call would be bound independently of where in the hierarchy the call is made. A healthy strategy can be obtained by restricting the binding to classes related to the class where the call is made. We let the notation $bind(D, C\#m)$ define the call $C\#m$ for search class D. The search is restricted by C; the returned class must be either above or below C. This ensures the healthiness condition described above. By type-safety, there is a definition of m above C; thus $bind(D, C\#m)$ is well-defined for D below C.

Definition 1. *Define* $bind(\_, \_\#\_) : List[Cid] \times Cid \times Mid \to Cid$ *by:*

$$\begin{array}{lll} bind(nil, C\#m) & \triangleq \bot & \\ bind(D\overline{D}, C\#m) & \triangleq D & \text{if } (D < C \vee D \geq C) \wedge m \in D.mtds \\ bind(D\overline{D}, C\#m) & \triangleq bind(D.inh\overline{D}, C\#m) & \text{if } (D < C \vee D \geq C) \wedge m \notin D.mtds \\ bind(D\overline{D}, C\#m) & \triangleq bind(\overline{D}, C\#m) & \text{otherwise} \end{array}$$

A *remote call* $x.m$ is bound by $bind(D, C\#m)$ where C is the declared class of x and D the actual class of x. A *statically bound* method call $m@C$ is bound above C independently of the actual class that the call is executed in. Following the traversal strategy above, the binding of the call is given by $bind(m@C)$:

Definition 2. *Define* $bind(\_@\_) : Mid \times Cid \to Cid$ *by:* $bind(m@C) \triangleq bind(C, C\#m)$.

Similar binding functions may be used to define the binding of fields: An occurrence of $f@B$ is allowed inside a class declaration C if B is above C, and is bound above B; and an unqualified occurrence of f inside C is understood as $f@C$.

3 Lazy Behavioral Subtyping

Lazy behavioral subtyping supports incremental reasoning for extensible class hierarchies; each class is analyzed based on the analysis of its superclasses, but independent of (future) subclasses. Lazy behavioral subtyping was presented for single inheritance in [13]. We here present an extension for multiple inheritance and horizontal name conflicts, assuming a healthy binding strategy. With healthy binding, a method call binds to a class related to the calling class. Therefore behavioral constraints may be propagated down the class hierarchy, which allows incremental reasoning. The proof method has two parts, a conventional program logic (e.g., [27,15,4,25]) and, on top of that, a *proof environment* which incrementally tracks method specifications and requirements.

The proof system uses Hoare triples $\{p\}\,t\,\{q\}$ with the standard partial correctness interpretation: if a statement t starts execution in a state where a precondition p holds and this execution terminates, then the postcondition q holds afterwards. Triples can be derived in any suited program logic, so let $\vdash_{\mathrm{PL}} \{p\}\,t\,\{q\}$ denote that the triple $\{p\}\,t\,\{q\}$ is derivable in the chosen program logic PL. A *proof outline* [25] for a method definition $m(\overline{x})\{t\}$ is an annotated method definition $m(\overline{x}) : (p,q)\,\{t\}$, where method calls inside t are decorated with call-site requirements. We henceforth assume that all method bodies are decorated in this way. The derivability $\vdash_{\mathrm{PL}} m(\overline{x}) : (p,q)\,\{t\}$ of a proof outline is given by $\vdash_{\mathrm{PL}} \{p\}\,t\,\{q\}$. Let *Spec* denote pairs (p,q) of conditions.

Method specifications and requirements. The verification technique distinguishes between a method's declared specification (its contract) and its *requirement*. Roughly, the first captures its announced behavior as declared in the pre- and post-condition of the method definition. In contrast, the requirements stem from call-sites and represent properties needed to verify the client code of a method, namely to satisfy the client code's specification in turn. Due to inheritance and overriding, a method with a given name is available in more than one class, and can be called from different client codes. Consequently, the properties are considered per class and its position in the class hierarchy. If, furthermore, the class hierarchy is incrementally extended, new specifications and requirements may be added. This bookkeeping of the properties is done in a proof environment, through the two mappings S and R.

Definition 3 (Proof environments). *A* proof environment *is a triple $\langle P, S, R\rangle$ of type Env, where P : Cid $\rightarrow$ Class is a partial mapping and R and S are total mappings of type Cid $\times$ Cid $\times$ Mid $\rightarrow$ Set[Spec].*

In such a proof environment $\mathcal{E}$, the mapping P reflects the class hierarchy and the two mappings S and R contain the constraints collected so far during analysis. We use a subscript, e.g., $R_{\mathcal{E}}$, if the proof environment is not clear from the context.

For a method m defined in a class B, besides m's declared specification as given in B itself, subclasses of B may give *additional* specifications for the method. For example, if a method n is overridden by a subclass C of B, and m calls n, a specification of $body(B,m)$ given by C may account for m's behavior relying on the overriding version of n. Hence, for a method m defined in B, $S(C,B.m)$ represents the specification as given in C. Note that a non-empty $S(C,B.m)$ implies $C \leq B$.

Example 2. Recall the method `update`, implemented in both `Account` and `Number` in Example 1. Let the specifications of these two definitions of `update` be contained in S(`Account`,`Account.update`) and S(`Number`,`Number.update`), respectively. The common subclass `InterestAccount` may provide *additional* specifications for these implementations in the sets S(`InterestAccount`,`Account.update`) and S(`InterestAccount`,`Number.update`).

In order to preserve a declared specification $(p,q) \in S(C,B.m)$ when inheriting m, it is necessary to impose requirements on methods called via late binding in $body(B,m)$. The requirements are given by the proof outline $m(\overline{x}) : (p,q)\,\{body(B,m)\}$ and maintained by the requirement mapping R. For each call $\{r\}\,n()\,\{s\}$ in this outline, the requirement (r,s) is included in $R(C,B\#n)$. Here, C denotes the class that *imposes* the requirement and B is the call-site class where m is defined.

Since we work with sets of specifications, the entailment relation is lifted as follows. Let p' be the condition p with all fields f substituted by f', avoiding name capture.

Definition 4 (Entailment). *Assume specifications (p,q) and (r,s), and specification sets $\mathcal{U} = \{(p_i,q_i) \mid 1 \leq i \leq n\}$ and $\mathcal{V} = \{(r_i,s_i) \mid 1 \leq i \leq m\}$.* Entailment *is defined by*

i) $(p,q) \rightarrow (r,s) \triangleq (\forall \overline{z}_1 \,.\, p \Rightarrow q') \Rightarrow (\forall \overline{z}_2 \,.\, r \Rightarrow s')$,
where $\overline{z}_1$ and $\overline{z}_2$ are the logical variables in (p,q) and (r,s), respectively
ii) $\mathcal{U} \rightarrow (r,s) \triangleq (\bigwedge_{1\leq i \leq n} (\forall \overline{z}_i \,.\, p_i \Rightarrow q_i')) \Rightarrow (\forall z \,.\, r \Rightarrow s')$.
iii) $\mathcal{U} \rightarrow \mathcal{V} \triangleq \bigwedge_{1\leq i \leq m} \mathcal{U} \rightarrow (r_i,s_i)$.

The relation $\mathcal{U} \rightarrow (r,s)$ corresponds to Hoare-style reasoning, proving $\{r\}\,t\,\{s\}$ from $\{p_i\}\,t\,\{q_i\}$ for all $1 \leq i \leq n$, by means of the adaptation and conjunction rules [3]. Entailment is reflexive and transitive, and $\mathcal{V} \subseteq \mathcal{U}$ implies $\mathcal{U} \rightarrow \mathcal{V}$.

Soundness. It is crucial for incremental reasoning to preserve the declared specifications for *inherited* methods: for a specification (p,q) included in $S(C,B.m)$ it is safe to rely on (p,q) when $body(B,m)$ is executed on an instance of subclasses of C. Note that *overriding* implementations of m in such subclasses may satisfy different contracts than the definition in the superclass. This flexibility goes beyond standard behavioral subtyping. With the open world assumption the subclasses of C are unknown when C is analyzed, so soundness is ensured by tracking the requirements that (p,q) imposes on late-bound calls in $body(B,m)$. If n is overridden in a class D below C, all requirements towards n made by classes above D must be satisfied by $body(D,n)$. This is expressed by $S(D,D.n) \rightarrow R\!\uparrow\!(D,n)$, where $R\!\uparrow\!(D,n)$ denotes the union of all requirements towards n made above D; i.e., the union of $R(C,B\#n)$ for all $D \leq C \leq B$.

In general soundness means that if $body(B,m)$ is executed on an instance of class D, it must be safe to rely on $S\!\uparrow\!(D,B.m)$, which is the union of $S(C,B.m)$ for all classes C where $D \leq C \leq B$. Soundness is formalized by the following definition of sound proof environments and Lemma 1. Let $C \in \mathcal{E}$ denote that $P_{\mathcal{E}}(C)$ is defined, and $x : C.m$ the remote call $x.m$ where x is declared with type C.

Definition 5 (Sound environments). *Let $B,C,D : Cid$ and $m,n : Mid$. A* sound environment *$\mathcal{E}$ satisfies the following two conditions for all $B,C \in \mathcal{E}$ and m:*

$$\begin{aligned} &i)\ \forall (p,q) \in S_{\mathcal{E}}(C,B.m)\ .\ \exists body(B,m)\ .\ \vdash_{\mathrm{PL}} m(\overline{x}) : (p,q)\, \{body(B,m)\} \\ &\quad \wedge Local_{\mathcal{E}}(C,B,body(B,m)) \wedge Rem_{\mathcal{E}}(body(B,m)) \wedge Stat_{\mathcal{E}}(C,body(B,m)) \\ &ii)\ m \in C.\mathrm{mtds} \Rightarrow S_{\mathcal{E}}(C,C.m) \rightarrowtriangle R{\uparrow}_{\mathcal{E}}(C,m) \end{aligned}$$

where

$$\begin{aligned} &Local_{\mathcal{E}}(C,B,t) \triangleq \forall \{r\}\, n\, \{s\} \in t\ .\ \forall D \leq_{\mathcal{E}} C\ .\ S{\uparrow}_{\mathcal{E}}(D, bind(D,B\#n).n) \rightarrowtriangle (r,s) \\ &Rem_{\mathcal{E}}(t) \triangleq \forall \{r\}\, x : D.n\, \{s\} \in t\ .\ S{\uparrow}_{\mathcal{E}}(D, bind(n@D).n) \rightarrowtriangle (r,s) \wedge\ R{\uparrow}_{\mathcal{E}}(D,n) \rightarrowtriangle (r,s) \\ &Stat_{\mathcal{E}}(C,t) \triangleq \forall \{r\}\, n@B\, \{s\} \in t\ .\ S{\uparrow}_{\mathcal{E}}(C, bind(n@B).n) \rightarrowtriangle (r,s) \end{aligned}$$

The soundness of a proof environment can be explained informally as follows: Assume that $(p,q) \in S_{\mathcal{E}}(C,B.m)$ and that there is a proof outline of $body(B,m)$ for (p,q). For each requirement $\{r\}\, n\, \{s\}$ to a *self-call* in this proof outline and for each subclass D of C, (r,s) must follow from the specifications of the method definition to which a call is bound for search class D. For each requirement $\{r\}\, x.n\, \{s\}$ to a *remote call*, (r,s) must follow from the specification of the method provided by the static type of x, and it must be imposed on redefinitions below the static type. For each requirement $\{r\}\, n@A\, \{s\}$ to a *static call*, (r,s) must follow from the specification of the method implementation to which the call will bind. The requirement is not imposed on method overridings since the call is bound at compile time.

Let $\models_C \{p\}\, t\, \{q\}$ denote $\models \{p\}\, t\, \{q\}$ provided that late-bound self-calls in t are bound for search class C, and let $\models_C m(\overline{x}) : (p,q)\, \{t\}$ be given by $\models_C \{p\}\, t\, \{q\}$. If t is without calls and $\vdash_{\mathrm{PL}} \{p\}\, t\, \{q\}$, then $\models \{p\}\, t\, \{q\}$ follows by the soundness of PL. Lemma 1 states that if $(p,q) \in S_{\mathcal{E}}(C,B.m)$ and $body(B,m)$ is executed in an instance of a subclass D of C, a sound environment guarantees that (p,q) is a valid specification:

Lemma 1. *Assume given a sound environment $\mathcal{E}$ and a sound program logic* PL. *Let $B,D : Cid$, $m : Mid$, and $(p,q) : Spec$ such that $B,D \in \mathcal{E}$ and $(p,q) \in S{\uparrow}_{\mathcal{E}}(D,B.m)$. Then $\models_D m(\overline{x}) : (p,q)\, \{body_{\mathcal{E}}(B,m)\}$.*

Example 3. Consider the method `Account.withdraw`(x), specified by $(\texttt{bal} = b_0, \texttt{bal} = b_0 - x) \in S(\texttt{Account}, \texttt{Account.withdraw})$. This specification leads to a requirement on `update`: the method modifies the balance according to its parameter. The requirement is satisfied by `update` defined in `Account`, and `FeeAccount`, the two implementations to which the call in `Account` can be bound. The separation of method specifications from requirements made by method calls allows incremental reasoning without imposing the constraints of behavioral subtyping on method overridings. For instance in `FeeAccount`, the overriding implementation satisfies $(\texttt{bal} = b_0, \texttt{bal} = b_0 - x - \texttt{fee@FeeAccount})$. Incremental reasoning is still supported as the static call to the superclass method relies on the verified specification of `withdraw` in `Account`. Correspondingly for the implementations of `withdraw` in `InterestAccount` and `Card`.

4 The Inference System for Incremental Reasoning

The inference system analyzes and manipulates the proof environments. Establishing a proof outline for one method at a given stage of the overall analysis gives rise to (further) proof-obligations, which are tracked by the proof system (cf. Section 4.1). The system itself is formalized as a set of derivation rules (cf. Section 4.3), whose traversal through the class-hierarchy is driven by the analysis operations given in Section 4.2.

4.1 Tracking Behavioral Constraints

Assume that a proof outline $m(\bar{x}) : (p,q)\,\{body(B,m)\}$ is given by a class C. To ensure soundness, this gives rise to the following steps:

1. (p,q) is included in $S(C,B.m)$.
2. for each call $\{r\}\,n\,\{s\}$ in the proof outline:
   (a) (r,s) is analyzed with regard to the implementation of $B\#n$ found for search class C; i.e., the proof obligation $S{\uparrow}(C,E.n) \rightharpoondown (r,s)$ must be established, where $E = bind(C,B\#n)$.
   (b) (r,s) is included in $R(C,B\#n)$.

Establishing $S{\uparrow}(C,E.n) \rightharpoondown (r,s)$ in step 2a means: Either (r,s) follows directly from the already established specifications in $S{\uparrow}(C,E.n)$ by entailment, or the proof outline $n(\bar{y}) : (r,s)\,\{body(E,n)\}$ given by C is analyzed in the same manner as the original specification of m. This adds (r,s) to $S(C,E.n)$, trivializing the proof of $S{\uparrow}(C,E.n) \rightharpoondown (r,s)$.

Including (r,s) into $R(C,B\#n)$ in step 2b constrains future subclasses of C: Each subclass D of C must ensure

$$S{\uparrow}(D,bind(D,B\#n).n) \rightharpoondown (r,s) \tag{1}$$

If n is overridden by D, all late-bound calls to n made by classes above D will bind to the definition of n in D. As explained, the calculus then ensures (1) by establishing $S(D,D.n) \rightharpoondown R{\uparrow}(D,n)$. If n is not overridden by D, we distinguish two cases. Let $E = bind(D,B\#n)$; i.e., the call $B\#n$ will bind to the implementation in E for search class D. If E is *related* to C, soundness of the analysis of superclasses of D ensures (1). If otherwise E is *unrelated* to C, class D may introduce a *diamond* in the class hierarchy, which needs to dealt with. A diamond is introduced by D if there are two different classes D_1 and D_2 in $D.inh$ and a class A such that $D_1 \leq A$ and $D_2 \leq A$. Let $commSup(D)$ denote the union of all such classes A. For an E unrelated to C, let $B \in commSup(D)$. Then the requirements $R(C,B\#n)$ were not imposed on $body(E,n)$ at the time E was analyzed. For soundness, they are therefore imposed on $body(E,n)$ when the diamond is created by D. More generally, the same argument applies to all classes between D and B that are unrelated to E. We let the set $dreq(D,B\#n)$ of *diamond requirements* denote the union of all $R(C,B\#n)$ for C such that $D \leq C \leq B$ and $bind(D,B\#n)$ is unrelated to C. By the analysis of D, the calculus ensures (1) by establishing $S{\uparrow}(D,E.n) \rightharpoondown dreq(D,B\#n)$.

Note that in the subcase where E is related to C, class D may also introduce a diamond. This case is covered by the proof of Theorem 1 (details are in the extended version [12]).

Example 4. In the classes of Example 1, the method `update` is defined in `Account` and overridden in `FeeAccount`. Let class `InterestAccount` impose a requirement (r,s) on `update`, contained in $R(\texttt{InterestAccount},\texttt{Account\#update})$. Now the class `Card` introduces a diamond in the class hierarchy. Since class `Card` is a subclass of `InterestAccount`, soundness requires the validity of the formula $S\!\uparrow\!(\texttt{Card}, \mathit{bind}(\texttt{Card},\texttt{Account\#update}).\texttt{update}) \rightarrowtriangle (r,s)$ where the method binding resolves to `FeeAccount`. Since `FeeAccount` and `InterestAccount` are unrelated, (r,s) is in the set $\mathit{dreq}(\texttt{Card},\texttt{Account\#update})$, and the calculus establishes the required verification of (r,s) by the analysis of `Card`.

4.2 Analysis Operations

The judgments of the calculus are of the form $\mathcal{E} \vdash \mathcal{A}$, where $\mathcal{E}$ is the proof environment and $\mathcal{A}$ is a list of *analysis operations* with the following syntax.

$$\begin{aligned}
&O ::= \varepsilon \mid \mathit{anMtd}(\overline{M}) \mid \mathit{anOutln}(C,t) \mid \mathit{verify}(C,m,\overline{R}) \mid \mathit{supCls}(\overline{C}) \mid \mathit{supMtd}(C,\overline{m}) \mid O \cdot O\\
&\mathcal{S} ::= \emptyset \mid L \mid \mathit{require}(C,m,(p,q)) \mid \mathcal{S} \cup \mathcal{S}\\
&\mathcal{A} ::= \mathit{module}(\overline{L}) \mid [\langle C:\ O\rangle\,;\mathcal{S}] \mid [\varepsilon\,;\mathcal{S}] \mid \mathit{module}(\overline{L}) \cdot \mathcal{A}
\end{aligned}$$

Here L denotes a class definition, as defined in Fig. 1. The rule system below specifies an algorithm that traverses a class hierarchy and its syntactic constituents — classes, methods, statements, etc. — according to the principles explained above; in particular, tracking specifications and requirements. The analysis starts with an $\mathcal{E} \vdash \mathcal{A}$ where $\mathcal{E}$ is empty and $\mathcal{A}$ contains the program as a sequence of modules. A module is a set of classes considered as a *compilation unit*. At each stage of the development, the modules given so far represent a complete, compilable program. Programs are open in the sense that new modules may be analyzed at later stages. Inside a module, the set $\mathcal{S}$ contains a module's classes. The inference rules ensure that a class can only be analyzed after analysis of all its superclasses.

The above operations and the proof environment drive the algorithm through the program. The operation **`class`** C **`extends`** $\overline{D}$ $\{\overline{f}\ \overline{M}\}$ initiates the analysis of C, and $[\langle C:\ O\rangle\,;\mathcal{S}]$ analyzes O in the context of class C *before* operations in $\mathcal{S}$ are considered. The analysis of a specific class involves the analysis of the proof outlines for its methods $\overline{M}$, the verification of the requirements for a method, and collecting the proof obligations for the calls mentioned inside the method bodies (by the operations $\mathit{anMtd}(\overline{M})$, $\mathit{verify}(D,m,\overline{R})$, and $\mathit{anOutln}(D,t)$). The operation $\mathit{require}(D,m,(p,q))$ applies to remote calls to ensure that m in D satisfies the requirement (p,q). Requirements are lifted outside the context of the analyzed class by this operation, and shifted into the set $\mathcal{S}$ of analysis operations. The two remaining operations, $\mathit{supCls}(\overline{D})$ and $\mathit{supMtd}(D,\overline{m})$ are only used during analysis of C, if C introduces diamonds.

Environment updates. Updates are represented by the operator $\_\oplus\_ : \mathit{Env} \times \mathit{Update} \rightarrow \mathit{Env}$, where the second argument represents the update. There are three different environment updates; loading a new class and extending the specifications or the requirements of a method in a class. The updates are defined as follows:

$$\begin{aligned}
&\mathcal{E} \oplus \mathit{extP}(C,\overline{D},\overline{f},\overline{M}) = \langle P_{\mathcal{E}}[C \mapsto \langle \overline{D},\overline{f},\overline{M}\rangle], S_{\mathcal{E}}, R_{\mathcal{E}}\rangle \\
&\mathcal{E} \oplus \mathit{extS}(C,D,m,(p,q)) = \langle P_{\mathcal{E}}, S_{\mathcal{E}}[(C,D,m) \mapsto S_{\mathcal{E}}(C,D,m) \cup \{(p,q)\}], R_{\mathcal{E}}\rangle \\
&\mathcal{E} \oplus \mathit{extR}(C,D,m,(p,q)) = \langle P_{\mathcal{E}}, S_{\mathcal{E}}, R_{\mathcal{E}}[(C,D,m) \mapsto R_{\mathcal{E}}(C,D,m) \cup \{(p,q)\}]\rangle
\end{aligned}$$

4.3 The Inference Rules

The inference rules are given in Fig. 4. Rule (NewModule) initiates the analysis of a set of classes. For convenience, we let $\overline{L}$ denote both a list and set of classes. Furthermore, (NewClass) loads a new class C for analysis, the second premise ensures that the superclasses $\overline{D}$ have already been analyzed. For each method m in C, the calculus generates an operation $\mathit{verify}(C,m,\overline{R})$, where $\overline{R}$ is the set of requirements that must hold for this method. Rules (ReqDer) and (ReqNotDer) deal with the verification of a particular specification with respect to the implementation. If the specification follows from the already established specification of the method, rule (ReqDer) continues with the remaining analysis operations. Otherwise, a proof of the specification is required. By (ReqNotDer), an outline of the specification is then analyzed by an *anOutln* operation. Remark that only rule (ReqNotDer) extends the S mapping.

For a given proof outline, the rules (LateCall), (StatCall), and (remCall) handle late-bound, static, and remote calls, respectively. Rule (LateCall) extends the R mapping and generates a *verify* operation to analyze the requirement for the implementation to which the call will bind. The extension of R ensures that the requirement will be respected by future subclasses. Rule (StatCall) also generates a *verify* operation, but does not extend R. Remote late-bound calls are handled by the rules (remReq) and (remCall), which allow reasoning from the method requirements given in the declared class of the callee. Notice that no new requirements are imposed. However, as requirements are generated from internal self-calls in a class, these may not provide suitable external specifications.

Finally, there are rules for analyzing requirements from common superclasses when diamonds are introduced in the environment. Rule (SupMtd) generates a *supMtd* for each common superclass. For each method called by a common superclass, (SupReq) generates a *verify* operation for the requirements imposed by calls to the method. If a class introduced by (NewClass) does not have any common superclasses, the generated *supCls* operation will have an empty argument and can be discarded by (NoSup).

For brevity, we elide a few straightforward rules which formalize a lifting from single-elements to sets or sequences of elements. For example, the rule for $\mathit{anMtd}(\overline{M})$ (which occurs in the premise of (NewClass)), generalizes the analysis of a single method which is done in (NewMtd). These rules are included in the extended version of this paper [12], together with the proof of the soundness theorem below. Note that a proof of $\mathcal{E} \vdash \mathit{module}(\overline{L})$ has exactly one leaf node $\mathcal{E}' \vdash [\varepsilon\,;\,\emptyset]$; we call $\mathcal{E}'$ the environment resulting from the analysis of $\mathit{module}(\overline{L})$.

Theorem 1. *Let $\mathcal{E}$ be a sound environment and $\overline{L}$ a set of class declarations. If a proof of $\mathcal{E} \vdash \mathit{module}(\overline{L})$ has $\mathcal{E}'$ as its resulting environment, then $\mathcal{E}'$ is also sound.*

(NewClass)

$$\frac{\begin{array}{c} C \notin \mathcal{E} \qquad \overline{D} \neq nil \Rightarrow \overline{D} \in \mathcal{E} \qquad \overline{E} = commSup_{\mathcal{E}}(C) \\ \mathcal{E} \oplus extP(C,\overline{D},\overline{f},\overline{M}) \vdash [\langle C : anMtd(\overline{M}) \cdot supCls(\overline{E})\rangle \,;\, \mathcal{S}] \cdot \mathcal{A} \end{array}}{\mathcal{E} \vdash [\varepsilon \,;\, \{\mathtt{\mathbf{class}}\ C\ \mathtt{\mathbf{extends}}\ \overline{D}\ \{\overline{f}\ \overline{M}\}\} \cup \mathcal{S}] \cdot \mathcal{A}}$$

(NewModule)

$$\frac{\mathcal{E} \vdash [\varepsilon \,;\, \overline{L}] \cdot \mathcal{A}}{\mathcal{E} \vdash module(\overline{L}) \cdot \mathcal{A}}$$

(NewMtd)

$$\frac{\mathcal{E} \vdash [\langle C : verify(C,m,\{(p,q)\} \cup R\uparrow_{\mathcal{E}}(C.inh,m)) \cdot \mathcal{O}\rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : anMtd(m(\overline{x}) : (p,q)\,\{t\}) \cdot \mathcal{O}\rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}$$

(EmpClass)

$$\frac{\mathcal{E} \vdash [\varepsilon \,;\, \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \varepsilon\rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}$$

(ReqDer)

$$\frac{S\uparrow_{\mathcal{E}}(C,D.m) \rightarrow (p,q) \qquad \mathcal{E} \vdash [\langle C : \mathcal{O}\rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : verify(D,m,(p,q)) \cdot \mathcal{O}\rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}$$

(EmpModule)

$$\frac{\mathcal{E} \vdash \mathcal{A}}{\mathcal{E} \vdash [\varepsilon \,;\, \emptyset] \cdot \mathcal{A}}$$

(ReqNotDer)

$$\frac{\begin{array}{c} \vdash_{\mathrm{PL}} m : (p,q)\,\{body_{\mathcal{E}}(D,m)\} \\ \mathcal{E} \oplus extS(C,D,m,(p,q)) \vdash [\langle C : anOutln(D, body_{\mathcal{E}}(D,m)) \cdot \mathcal{O}\rangle \,;\, \mathcal{S}] \cdot \mathcal{A} \end{array}}{\mathcal{E} \vdash [\langle C : verify(D,m,(p,q)) \cdot \mathcal{O}\rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}$$

(LateCall)

$$\frac{E = bind(C, D\#m) \qquad \mathcal{E} \oplus extR(C,D,m,(p,q)) \vdash [\langle C : verify(E,m,(p,q)) \cdot \mathcal{O}\rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : anOutln(D, \{p\}\,m\,\{q\}) \cdot \mathcal{O}\rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}$$

(StatCall)

$$\frac{\mathcal{E} \vdash [\langle C : verify(bind(m@B),m,(p,q)) \cdot \mathcal{O}\rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : anOutln(D, \{p\}\,m@B\,\{q\}) \cdot \mathcal{O}\rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}$$

(remCall)

$$\frac{\mathcal{E} \vdash [\langle C : \mathcal{O}\rangle \,;\, \mathcal{S} \cup \{require(E,m,(p,q))\}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : anOutln(D, \{p\}\,x : E.m\,\{q\}) \cdot \mathcal{O}\rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}$$

(remReq)

$$\frac{C \in \mathcal{E} \qquad R\uparrow_{\mathcal{E}}(C,m) \rightarrow (p,q) \qquad S\uparrow_{\mathcal{E}}(C, bind(m@C).m) \rightarrow (p,q) \qquad \mathcal{E} \vdash [\varepsilon \,;\, \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\varepsilon \,;\, \{require(C,m,(p,q))\} \cup \mathcal{S}] \cdot \mathcal{A}}$$

(SupMtd)

$$\frac{\mathcal{E} \vdash [\langle C : supMtd(D, called_{\mathcal{E}}(D) \setminus C.mtds) \cdot \mathcal{O}\rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : supCls(D) \cdot \mathcal{O}\rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}$$

(SupReq)

$$\frac{E = bind(C, D\#m) \qquad \mathcal{E} \vdash [\langle C : verify(E,m,dreq(C,D\#m)) \cdot \mathcal{O}\rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : supMtd(D,m) \cdot \mathcal{O}\rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}$$

(NoSup)

$$\frac{\mathcal{E} \vdash [\langle C : \mathcal{O}\rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : supCls(\emptyset) \cdot \mathcal{O}\rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}$$

(NoSupMtd)

$$\frac{\mathcal{E} \vdash [\langle C : \mathcal{O}\rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : supMtd(D,\emptyset) \cdot \mathcal{O}\rangle \,;\, \mathcal{S}] \cdot \mathcal{A}}$$

Fig. 4. The inference system. Here m denotes a call, including actual parameters, and $called(D)$ denotes the names of the methods called by D.

5 Methodological Aspects

With the given approach, a programmer typically provides S-requirements for each class. Their verification generates R-requirements for the late-bound self-calls occurring in the class, which will be imposed on subclass redefinitions of the called methods. In a subclass C, redefined methods can violate the S-requirements of a superclass, but not the R-requirements. C may give additional contracts for inherited methods, resulting in additional verification of such methods, which may generate additional R-requirement for future subclasses of C. With multiple inheritance, this means that different parts of the inheritance graph may have different R-requirements to the same method. Note that behavioral subtyping is not implied by this approach: When m is overridden, the new definition need not implement all superclass specifications of m, but only the R-requirements made towards usage of m. This way, lazy behavioral subtyping still supports incremental reasoning under an open world assumption.

A weakness of the approach presented here is that a remote call $x.m(..)$ may create R-requirements to m for the declared class of x, say C, and these requirements must be imposed on C and its subclasses, unless they follow from already established R-requirements to m for C. Adding R-requirements to a previously established class hierarchy can lead to several verification tasks, which makes the approach less modular. As R-requirements generated from internal self-calls in a class may not in general provide suitable external properties, a programmer should provide R-requirements such that reasoning about remote calls can be derived from these. Therefore a programmer should be aware of the distinction between S- and R-requirements, and be able to provide both, and that unnecessarily strong R-requirements will restrict future method redefinitions.

A more modular version of lazy behavioral subtyping may be obtained by using *behavioral interfaces*. A behavioral interface describes the visible methods of a class and their contracts (or possibly an invariant), and inheritance may be used to form new interfaces from old ones. An advantage of seeing all classes through interfaces is that explicit hiding constructs become superfluous. A class may then be specified by a number of interfaces. If all object variables (references) are typed by interfaces, one may let the inheritance hierarchies of interfaces and classes be independent. In particular, one need not require that a subclass of C inherits (nor respects) the behavioral interfaces specified for C: Static type checking of an assignment $x := e$ must then ensure that the expression e denotes an object supporting the declared interface of the object variable x. In this setting, the substitution principle for objects can be reformulated as follows: *For an object variable x with declared interface I, the actual object referred to by x at run-time will satisfy the behavioral specification I.* As a consequence, a subclass may freely reuse and redefine superclass methods, since it is free to violate the behavioral specification of superclasses. Reasoning about a remote call $x.m(..)$ can then be done by relying on the behavioral interface of the object variable x, simplifying rule (REMREQ) to simply check interface contracts. This approach is followed by, e.g., Creol [18].

6 Related Work

Multiple inheritance is supported in, e.g., C++ [31], CLOS [11], Eiffel [23], POOL [2], and Self [9]. Horizontal name conflicts in C++, POOL, and Eiffel are removed by explicit resolution, after which the inheritance graph may be linearized. Multiple dispatch, or multi-methods [11], gives a more powerful binding mechanism, but reasoning about multi-methods and redefinition is difficult. The prototype-based language Self [9] proposes an elegant *prioritized binding strategy*. Each superclass is given a priority. With equal priority, the superclass related to the caller class is preferred. However, explicit class priorities may cause surprises in large class hierarchies: names may become ambiguous through inheritance. If neither class is related to the caller, binding fails.

Formalizations of multiple inheritance in the literature traditionally use the *objects-as-records* paradigm. This approach addresses subtyping issues related to subclassing, but method binding is not easily captured. In Cardelli's denotational semantics of multiple inheritance [8], not even access to methods of superclasses is addressed. Rossie, Friedman, and Wand [29] formalize multiple inheritance using *subobjects*, a run-time data structure used for virtual pointer tables [19,31]. This work focuses on compile-time issues and does not clarify multiple inheritance at the abstraction level of the programming language. A natural semantics for late binding in Eiffel models the binding mechanism at the abstraction level of the program [5]. Recently, an operational semantics and type safety proof inspired by C++ has been formalized in Isabelle [32].

Work on behavioral reasoning about object-oriented programs address languages with single inheritance (e.g., [27, 28, 7]). For late binding, different variations of behavioral subtyping are most common [21, 1, 20], as discussed above. Pierik and de Boer [27] present a sound and complete reasoning system for late-bound calls which does not rely on behavioral subtyping. This work, also for single inheritance, is based on a closed world assumption, meaning that the class hierarchy is not open for incremental extensions. To support object-oriented design, proof systems should be constructed for incremental reasoning.

Lately, incremental reasoning, both for single and multiple inheritance, has been considered in the setting of *separation logic* [22, 10, 26]. These approaches support a distinction between static specifications, given for each method implementation, and dynamic specifications that are used to verify late-bound calls. The dynamic specifications are given at the declaration site, in contrast to our work where late-bound calls are verified based on call-site requirements.

7 Conclusion and Future Work

Lazy behavioral subtyping supports incremental reasoning under an open world assumption, where class hierarchies can be gradually extended by inheritance. The approach is more flexible than traditional behavioral subtyping, as illustrated by the running example. This paper has introduced a healthiness condition for method binding and extended lazy behavioral subtyping to the setting of multiple inheritance for healthy binding strategies. This extension requires additional context information for method specifications and requirements, in order to resolve ambiguities that do not occur in single inheritance languages. The combination of healthiness and lazy behavioral subtyping has the advantage that requirements from two independent class hierarchies do not interfere with each other when the hierarchies are combined in a common subclass. This is essential in an incremental proof system.

The inference rules for incremental reasoning presented in this paper are essentially syntax-driven and would form a good basis for integrating behavioral reasoning in a tool supported environment for program development. In such a tool, specifications for method definitions must be manually annotated, whereas method requirements in proof outlines may often be inferred. The integration of lazy behavioral subtyping in the KeY tool [6] is currently being investigated. This integration will allow more elaborate case studies to better evaluate the methodology and practical applicability of the approach.

References

1. America, P.: Designing an object-oriented programming language with behavioural subtyping. In: de Bakker, J.W., de Roever, W.-P., Rozenberg, G. (eds.) REX 1990. LNCS, vol. 489, pp. 60–90. Springer, Heidelberg (1991)
2. America, P., van der Linden, F.: A parallel object-oriented language with inheritance and subtyping. In: Meyrowitz, N. (ed.) Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 1990), vol. 25(10), pp. 161–168. ACM Press, New York (1990)
3. Apt, K.R.: Ten years of Hoare's logic: A survey — Part I. ACM Transactions on Programming Languages and Systems 3(4), 431–483 (1981)
4. Apt, K.R., Olderog, E.-R.: Verification of Sequential and Concurrent Systems. In: Texts and Monographs in Computer Science. Springer, Heidelberg (1991)
5. Attali, I., Caromel, D., Ehmety, S.O.: A natural semantics for Eiffel dynamic binding. ACM Transactions on Programming Languages and Systems 18(6), 711–729 (1996)
6. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
7. Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer 7(3), 212–232 (2005)
8. Cardelli, L.: A semantics of multiple inheritance. Information and Computation 76(2-3), 138–164 (1988)
9. Chambers, C., Ungar, D., Chang, B.-W., Hölzle, U.: Parents are shared parts of objects: Inheritance and encapsulation in SELF. Lisp and Symbolic Computation 4(3), 207–222 (1991)
10. Chin, W.-N., David, C., Nguyen, H.-H., Qin, S.: Enhancing modular OO verification with separation logic. In: Necula and Wadler [24], pp. 87–99.
11. Demichiel, L.G., Gabriel, R.P.: The common lisp object system: An overview. In: Bézivin, J., Hullot, J.-M., Lieberman, H., Cointe, P. (eds.) ECOOP 1987. LNCS, vol. 276, pp. 151–170. Springer, Heidelberg (1987)
12. Dovland, J., Johnsen, E.B., Owe, O., Steffen, M.: Incremental reasoning for multiple inheritance. Research Report 373, Dept. of Informatics, University of Oslo (April 2008), `http://heim.ifi.uio.no/~creol`
13. Dovland, J., Johnsen, E.B., Owe, O., Steffen, M.: Lazy behavioral subtyping. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 52–67. Springer, Heidelberg (2008)
14. Fournet, C., Laneve, C., Maranget, L., Rémy, D.: Inheritance in the Join calculus. Journal of Logic and Algebraic Programming 57(1-2), 23–69 (2003)
15. Hoare, C.A.R.: An Axiomatic Basis of Computer Programming. Communications of the ACM 12, 576–580 (1969)
16. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. ACM Transactions on Programming Languages and Systems 23(3), 396–450 (2001)

17. Johnsen, E.B., Owe, O.: A dynamic binding strategy for multiple inheritance and asynchronously communicating objects. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2004. LNCS, vol. 3657, pp. 274–295. Springer, Heidelberg (2005)
18. Johnsen, E.B., Owe, O., Yu, I.C.: Creol: A type-safe object-oriented model for distributed concurrent systems. Theoretical Computer Science 365(1–2), 23–66 (2006)
19. Krogdahl, S.: Multiple inheritance in Simula-like languages. BIT 25(2), 318–326 (1985)
20. Leavens, G.T., Naumann, D.A.: Behavioral subtyping, specification inheritance, and modular reasoning. Tech. Rep. 06-20a, Dept. of Comp. Sci., Iowa State University (2006)
21. Liskov, B.H., Wing, J.M.: A behavioral notion of subtyping. ACM Transactions on Programming Languages and Systems 16(6), 1811–1841 (1994)
22. Luo, C., Qin, S.: Separation logic for multiple inheritance. Electronic Notes in Theoretical Computer Science 212, 27–40 (2008)
23. Meyer, B.: Object-Oriented Software Construction, 2nd edn. Prentice-Hall, Englewood Cliffs (1997)
24. Necula, G.C., Wadler, P. (eds.): 37th Annual Symposium on Principles of Programming Languages (POPL 2008). ACM Press, New York (2008)
25. Owicki, S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Informatica 6(4), 319–340 (1976)
26. Parkinson, M.J., Biermann, G.M.: Separation logic, abstraction, and inheritance. In: Necula and Wadler [24]
27. Pierik, C., de Boer, F.S.: A proof outline logic for object-oriented programming. Theoretical Computer Science 343(3), 413–442 (2005)
28. Poetzsch-Heffter, A., Müller, P.: A programming logic for sequential Java. In: Swierstra, S.D. (ed.) ESOP 1999. LNCS, vol. 1576, pp. 162–176. Springer, Heidelberg (1999)
29. Rossie Jr., J.G., Friedman, D.P., Wand, M.: Modeling subobject-based inheritance. In: Cointe, P. (ed.) ECOOP 1996. LNCS, vol. 1098, pp. 248–274. Springer, Heidelberg (1996)
30. Soundarajan, N., Fridella, S.: Inheritance: From code reuse to reasoning reuse. In: Devanbu, P., Poulin, J. (eds.) Proc. Fifth International Conference on Software Reuse (ICSR5), pp. 206–215. IEEE Computer Society Press, Los Alamitos (1998)
31. Stroustrup, B.: Multiple inheritance for C++. Computing Systems 2(4), 367–395 (1989)
32. Wasserrab, D., Nipkow, T., Snelting, G., Tip, F.: An operational semantics and type safety proof for multiple inheritance in C++. In: Tarr, P.L., Cook, W.R. (eds.) Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2006), pp. 345–362. ACM, New York (2006)

Model Checking LTL Formulae in RAISE with FDR

Abigail Parisaca Vargas[1], Ana G. Garis[2], S. Lizeth Tapia Tarifa[1], and Chris George[3]

[1] San Pablo Catholic University, Arequipa, Peru
[2] University of San Luis, Argentina
[3] International Institute for Software Technology, United Nations University, Macao

Abstract. The Raise Specification Language (RSL) is a modeling language which supports various specification styles. To apply model checking to RSL concurrent descriptions, we translate RSL specifications into the input language CSPM of FDR. FDR is the model checker for the process algebra CSP. First, we define a syntactic and semantic translation from the concurrent applicative subset of RSL to CSPM, and show that this translation is a strong bisimulation which preserves properties such as traces and deadlock. Consequently, results obtained by refinement checks in FDR are sound for the original RSL descriptions. Second, RSL uses Linear Temporal Logic (LTL) to specify desired properties, but FDR does not support LTL. LTL formulas may be translated to CSP test processes in order to check them with FDR. We build a tool which automates the translation of RSL specifications into CSPM and translates LTL formulas to CSP processes, enabling the model checking of LTL formulas over RSL descriptions with FDR.

Keywords: RAISE, RSL, CSP, FDR, formal methods, model checking, refinement, tools, LTL.

1 Introduction

Concurrent systems are increasingly necessary in society. Two characteristics that make concurrent processes difficult to understand are distribution and reactivity. In order to facilitate the modeling and verification of such complicated systems, we need powerful languages and tools than can facilitate the modeling and verification of them.

Although many different kinds of modelling languages and tools are available, it can still be difficult to model and reason about these systems. Combining the use of two or more of these formalisms may be suitable to model a particular system. Moreover, it is also possible to integrate different formalisms to automate the different kinds of tasks that should be done.

In this paper, we describe the steps we followed in order to model check Raise Specification Language (RSL) descriptions using the CSP model checker FDR [10]. Section 2 discusses model checking in the context of RSL.

In Section 3 we describe our approach to develop a translation from RSL [5] to CSP [6,16,14], in particular to its machine-readable variant CSP_M. We establish a concurrent applicative subset of RSL that has a translation to CSP_M, then we establish a semantic link between an RSL specification and its CSP_M translation by means of bisimulation. This translation is, actually, a strong bisimulation that preserves properties such as deadlock and divergence. Hence we show that results obtained by refinement checks in FDR are sound with respect to the original RSL description.

In Section 4 we consider how to assert and prove properties of our models. In model checking by refinement, one has to express the required property in terms of a more abstract process: the property is valid for a model if the property's description as a process is refined by the model. In CSP there are three kind of refinement: traces, failures and failures-divergences. Each one is specially useful for proving different properties; for instance, traces refinement is useful for proving safety properties. It is not always convenient to express properties in terms of more abstract processes, and temporal logic has been the traditional means of stating properties to be model checked.

The relationship between the refinement-based and temporal logic approach was studied by Leuschel, Massart and Currie [9,8]. They show an approach for doing LTL model checking of CSP specifications using refinement checking in FDR. They present a way to handle deadlocking systems, and discuss the validity for infinite state systems. After analyzing Leuschel, Massart and Currie's approach, we take the general idea and adapt it to translate LTL formulae from RSL to CSP processes.

Section 5 gives a short description of our tool, which implements the various steps of the approach explained in the paper.

Section 6 summarizes the achievement: a tool that translates a subset of RSL into CSP_M in order to apply refinement checking techniques over RSL specifications using FDR, and the translation of LTL formulae from RSL descriptions to CSP tester processes, so we can apply the traditional model checking technique to applicative concurrent RSL specifications.

2 Model Checking

Model checking is an automatic technique for verifying finite state systems. Enabling model checking often requires building a smaller, more abstract or simplified model of the main design that preserves its essential characteristics but avoids complexity. The idea then is to verify this model.

Applying model checking to a model includes mainly these three tasks: modeling (construction of the model), specification (definition of the properties to be checked) and verification (mechanical checking of the properties against the model). Many formal languages have been used for modeling, with corresponding tools for verification. Some form of temporal logic is the most common language for expressing properties, and the most useful for asynchronous processes is probably LTL [12], which we use.

2.1 Traditional Model Checking and Model Checking by Refinement

The traditional model checking (MC) method is a verification process which decides whether p holds for M ($M, s \models p$); where M is a model structure with a initial state s and p is a desired property of the system. LTL formulae allow one to specify different requirements of systems; in particular, they are useful to correctly specify safety and liveness properties.

The FDR tool is a refinement checker for CSP models and it supports three models: traces model, failures model and failures/divergences model [10]. *Traces(P)* represents the set of finite sequences of communications that a process P can perform. *Failures(P)* represents the set of all the pairs (s, X), where ‘s’ is a finite trace of a process P and ‘X’ is a set of refusals, the events that P can refuse to participate in after doing the trace s. *Divergences(P)* represents the set of failures and also the set of divergences; while or after a divergence happens, P can perform an infinite sequence of consecutive internal actions. The forms of refinement corresponding to the traces, failures and divergences models are called *traces refinement*,*failures refinement* and *failures-divergences refinement* respectively. Traces refinement allows the checking for safety properties and failures refinement the checking of deadlock freedom, while failures-divergences refinement is the most appropriate for checking livelock freedom.

2.2 Model Checking in RSL

RAISE is a formal method, with RSL as its specification language. A set of tools [4] is available for RSL, including test coverage analysis and mutation testing, translators to different languages, and a translator to PVS which allows RSL specifications to be proved by the PVS theorem prover. Model checking has been supported using the Symbolic Analysis Laboratory (SAL) as a third party model checker. This also involved adding the possibility to include LTL assertions in RSL specifications.

2.3 Using FDR for RSL

The SAL tool in the RAISE suite does not allow one to model check concurrent RSL specifications. Our purpose then was to translate the RSL applicative concurrent style process models to suitable CSP ones in order to apply tools like FDR which can help us to model RSL processes. This raises two issues

1. Translation: There are syntactic and semantic differences between RSL and CSP which we need to cope with.
2. Specification: We need to find a way to translate LTL into CSP processes.

3 Translation

We want to translate RSL applicative concurrent descriptions to corresponding CSP ones. We describe the translation in syntactic terms, and then establish formally what we mean by “corresponding”: we will show that the RSL and CSP descriptions are strongly bisimilar. The details are in a technical report [18]: we give just an overview here.

3.1 The Syntax

Since we are interested in the translation of RSL to CSP_M, the variant of CSP accepted as input by FDR, we need to identify the features of RSL that can be expressed in CSP_M. This translation subset includes:

- The built-in types **Bool**, **Int** and **Nat** as shown in Table 1.
- The compound types **Product**, **Set**, **List**, **Subtype**, **Variant** and **Record** as shown in Table 2. Since complex variants and records in RSL provide an implicit constructor, destructors, and (optionally) reconstructors, equivalent functions to these features have to be created during the translation to CSP_M.

Table 1. Built-in types translation from RSL to CSP_M

Language	Type	Values	Operators
RSL	**Bool**	**true**, **false**	$\wedge, \vee, \sim, =, \neq$
CSP_M	Boolean	$true, false$	$\wedge, \vee, \neg$, $==, !=$
RSL	**Int**	...,-1,0,1...	$+, -, *, /, \backslash, <, \leq, >, \geq$
CSP_M	Number	...,-1,0,1...	$+, -, *, /, \%, <, \leq, >, \geq$
RSL	**Nat**	0,1...	$+, -, *, /, \backslash, <, \leq, >, \geq$
CSP_M	Open range set	{0..}	$+, -, *, /, \%, <, \leq, >, \geq$

Table 2. Compound types translation from RSL to CSP_M

Language	Type	Example definition	Example value	Operators
RSL	**Product**	**type** Position = **Int** × **Int**	(-6,5)	
CSP_M	Tuple		(-6,5)	
RSL	**Set**	**type** IntegerSet = **Int-set**	{1..5}	$\cup, \cap, \backslash, \in$, **card**
CSP_M	Set		{1..5}	$\cup, \cap, -, \in$, *card*
RSL	**List**	**type** IntegerList = **Int***	$\langle 1, 2, 3\rangle$	**hd**, **tl** , $^\frown$, **len**
CSP_M	Sequence		$\langle 1, 2, 3\rangle$	*head*, *tail*, $^\frown$, #
RSL	**Subtype**	**type** Gun = {| i : **Nat** • i ∈ {0..15} |}		
CSP_M	Close range set	$nametype\ Gun = \{0..15\}$		
RSL	**Variant**	**type** ComplexColour == RGB(Red : Gun, Green : Gun, Blue : Gun) | Black | White		
CSP_M	Data type	$datatype\ ComplexColour = RGB.Gun.Gun.Gun \mid Black \mid White$ $Red(RGB.vRed.vGreen.vBlue) = vRed$ $Green(RGB.vRed.vGreen.vBlue) = vGreen$ $Blue(RGB.vRed.vGreen.vBlue) = vBlue$		
RSL	**Record**	**type** Figure :: S : Shape C : ComplexColour		
CSP_M	Data type	$datatype\ Figure = mk\_Figure.Shape.ComplexColour$ $S(mk\_Figure.vS.vC) = vS$ $C(mk\_Figure.vS.vC) = vC$		

- Explicit constant values of built-in types and compound types.
- Simple channels and channel arrays as shown in Table 3.
  In RSL a channel array is expressed through an object array and it is translated to a complex protocol channel in CSP_M.
- Explicit function and process with **if**, **case** and/or **let** expressions as shown in Table 4. CSP_M does not support elsif and case process expressions, but they can be simulated using If and Let expressions in the translation to CSP_M.

Table 3. Channel translation from RSL to CSP_M

Channel	**Language**	**Example definition**
Simple channel	RSL	**channel** mess : Index × Data
	CSP_M	$channel\ mess : (Index, Data)$
Channel array	RSL	**object** fork[p : Index, f : Index] : **class channel** pickup, putdown : **Unit end**
	CSP_M	$channel\ fork\_pickup, fork\_putdown : Index.Index$

Table 4. Function and process expression translation from RSL to CSP_M

Expression	**Language**	**Example**
If	RSL	**if** x > y **then** x − y **else** y − x **end**
	CSP_M	if $x > y$ then $x - y$ else $y - x$
Let	RSL	**let** p = input? **in let** (x,y) = p **in** output!x+y; PROC_PLUS() **end end**
	CSP_M	$input?p \rightarrow$ let $(x, y) = p$ within $output!x + y \rightarrow PROC\_PLUS$
Case	RSL	**case** p **of** (**true**,**false**) → **true**, _ → **false** **end**
	CSP_M	$if\ let\ (x1\_, x2\_)\ =\ p$ $within\ x1\_\ ==\ true\ and\ x2\_\ ==\ false$ $then\ let\ (x1\_, x2\_)\ =\ p$ $within\ true\ else\ false$

– The communication primitives sequence, internal and external choice, comprehended internal and external choice, parallel and comprehended parallel; and the basic processes **stop** and **skip**. To find an equivalence between these communication primitives and basic processes in RSL and CSP_M, it is necessary to evaluate the operational semantics of these two languages.

3.2 The Semantics

In this section we compare the semantics of the subsets of RSL and CSP.

The operators of RSL and CSP. Based on the grammar of the process expression in both process algebras, we show the operators which can be translated in both cases.

The RSL expressions are as follows:

$P =$ **skip** | **stop** | $E;\ P \mid P \sqcap P \mid P \Box P \mid P \| P \mid$ **if** v **then** P **else** P **end**
$E =$ c? | c!v
where v is a pure value expression, and P is a process expression.

The CSP expressions are as follows:

$P = SKIP \mid STOP \mid E \rightarrow P \mid P \sqcap P \mid P \Box P \mid P \| P \mid if\ v\ then\ P\ else\ P$
$E = c? \mid c!v$
where v is a pure value expression, and P is a process expression.

Although the operators are syntactically similar, they are sometimes semantically different as discussed below.

Operational Semantics Comparison. The operational semantics rules are taken from [1] in the case of the RSL rules and from [16] in the case of the CSP rules.

We do not include the details here, but simply state that the internal and external choice combinators have equivalent semantics in the two languages. The differences lie in parallelism, essentially since CSP adopts a "broadcast" semantics to communication between parallel processes, while RSL adopts a "point-to-point" semantics.

We consider two cases for parallel processes : synchronization and non-synchronization.

Synchronization. The operational semantics rules for synchronization of RSL and CSP are the following:

RSL	CSP
$\dfrac{\rho \vdash \mathrm{P} \xrightarrow{a} \mathrm{P}', \quad \mathrm{Q} \xrightarrow{\bar{a}} \mathrm{Q}'}{\rho \vdash \mathrm{P} \Vert \mathrm{Q} \xrightarrow{\tau} \mathrm{P}' \Vert \mathrm{Q}'}$ (1)	$\dfrac{P \xrightarrow{a} P', \quad Q \xrightarrow{a} Q'}{P \Vert Q \xrightarrow{a} P' \Vert Q'}$ (2)

where if a is an input (c?x) then $\bar{a}$ is an output on the same channel (c!v), and vice versa. Note that in RSL the event is "consumed" by the synchronization, becoming a τ. The CSP rule is more general, in that both events may be inputs or outputs. The event in CSP is not consumed; other processes running in parallel may also participate in it. We see that the two rules will coincide only for matched inputs and outputs, and provided we hide synchronized events as they occur.

To deal with this problem, we adopt the following as a *design rule*:

A process can only either input or output on a channel, and at most one other process can access that channel, and the access is in the opposite direction.

This rule may seem restrictive, but is in fact advised by the RAISE Method [13], and is natural in a language with point-to-point communication. If we adopt this rule for RSL then we see that such RSL translated in the natural manner to CSP will produce CSP processes in which the two sub cases where both events are inputs, or both are outputs, cannot occur.

This rule is statically checkable provided there are no channel arrays.

Non-synchronization. A non-synchronised transition between parallel processes occurs when one process makes a transition and the other does not. The relevant semantics rules for CSP and RSL show that for an internal event the rules are the same. But for a visible event CSP alone requires that the event involved in the transition of one process is not in the alphabet of the other. There is no such restriction in RSL. This is necessary in a language like RSL with point-to-point communication to preserve the associativity of the parallel operator. Suppose, for example, that P can output on channel c, and both Q and R can input on c. Then the combination (P || Q) || R must be able to progress by

P communicating with either Q or R, and for P to communicate with R, (P || Q) must be able to output on c, without Q being involved.

We can see that our design rule takes care of this problem by not allowing such a parallel combination: we cannot have both Q and R inputting on channel c.[1]

So we adopt the following rule for translating parallel processes:

$$(P \| Q)_T = (P_T \| Q_T) \backslash \alpha P_T \cap \alpha Q_T$$

where X_T is the CSP process translated from the RSL process X.

Soundness. Soundness means establishing that the results obtained from tools applied to the CSP model are valid for the original RSL. So we need to establish the following proof rule:

$$\frac{R_T \models P_T}{R \vdash P}$$

where we want to prove property P of RSL specification R, translated to P_T and R_T respectively.

FDR is essentially a refinement checker, so P_T is typically a (traces, failures, or failures-divergence) refinement relation, but may also be an assertion of deadlock freedom. We can therefore establish soundness by establishing the following:

1. that the translation scheme is a strong bisimulation. We need strong rather than weak bisimilarity to include divergence as a property that is preserved.
2. that strongly bisimilar processes have the same traces, failures, divergences and deadlocks.

The details are in a technical report [20]: we give a brief summary here.

Bisimulation. We first establish the obvious mapping of the events and basic processes of RSL and CSP. We then proceed by structural induction over the syntax of RSL processes: for each construction we assume the component processes are bisimilar to their translations, and show the constructed process to be bisimilar to its translation.

For example, to prove bisimilarity for the parallel combinator, we assume P is bisimilar to its translation P_T, and Q is bisimilar to its translation Q_T. We then show that $P \| Q$ is bisimilar to its translation (given above): call this PQ_T. To show bisimilarity we consider each possible transition of $P \| Q$ according to the RSL operational semantics to process X, say, and show there is a corresponding transition for PQ_T in the *CSP* operational semantics to a process which is bisimilar to X. Then we do the converse, considering each possible transition of PQ_T, finding a corresponding transition for $P \| Q$, and showing the resulting processes are bisimilar.

[1] We can also see that our design rule does not remove the possibility of such an architecture. We replace c by two channels cpq and cpr, say, so that Q inputs on cpq and R on cpr, and P makes an internal (non-deterministic) output choice between cpq and cpr. This is semantically equivalent to the original system (assuming no other processes access c) and obeys our design rule.

Properties. We prove the following for a process P and its (strongly bisimilar) translation P_T:

1. Traces: P can do a trace l iff P_T can do a trace l.
2. Deadlock: P can deadlock iff P_T can deadlock.
3. Refusals: x is a refusal of P iff x is a refusal of P_T
4. Failures: $failures$(P) = $failures(P_T)$.
5. Divergences: P diverges iff P_T diverges.

This means that properties we can prove of translated CSP scripts using FDR (lack of deadlock, trace-, failures- or failures-divergence-refinement) must also be true of the original RSL descriptions. In other words we have shown that FDR is a *sound* model checker for applicative concurrent RSL descriptions.

4 Specification

In this section, we deal with the modelling of LTL formulae. We first explain how LTL formulae may be modelled as tester processes in CSP, following the approach of Leuschel, Massart, and Currie [9,8]. We then discuss how this approach can be adapted to the RSL setting.

4.1 A Translation of LTL Formulae to CSP

After a careful study of the relationship between the refinement-based approach and temporal logic, Leuschel, Massart and Currie [9,8] propose to make a general solution building a tester for each possible LTL formula. Looking at the possible LTL formulae they observe that in general infinite traces have to be tested in order to infer whether a formula is satisfied. They check the satisfaction of a LTL property following the procedure defined by Vardi and Wolper [19]; that is, verifying that $[[S]]_w \cap [[\neg\phi]]_w = \emptyset$ (where $[[S]]_w$ represents all the traces of the system$'$s model and $[[\neg\phi]]_w$ all the traces of the negation of a LTL formula). If the intersection between $[[S]]_w$ and $[[\neg\phi]]_w$ is empty then $S \models \phi$.

The approach consists of building a tester T_ϕ from a formula ϕ, composing T_ϕ with a system S, and checking if the composition satisfies some property. T_ϕ is built by translating ϕ to the corresponding Büchi automaton and translating this automaton to CSP; finally, FDR is used for checking emptiness. Special attention is paid to deadlocking traces, so they build a tester T_ϕ which accepts infinite traces and also deadlocking traces.

Deadlocking treatment and tester building. An extended LTL called LTL$_\Delta$ is defined in order to handle deadlocking traces. LTL$_\Delta$ is specified in the same way as LTL but over an extended alphabet. That is, while LTL is defined over Σ, LTL$_\Delta$ is over $\Sigma \cup \{\Delta\}$; where $\Delta \notin \Sigma$. If a valid trace π is finite then it is over Σ terminating on infinite Δ's; otherwise π is an infinite trace over Σ. Regarding the semantics of LTL$_\Delta$, two rules are defined in [9,8] for translating LTL into LTL$_\Delta$:

1) $X\phi \quad \rightsquigarrow \quad \neg\Delta \wedge X\phi$
2) $\neg X\phi \quad \rightsquigarrow \quad \Delta \vee X\neg\phi$

Also, the definition of when an LTL$_\Delta$ formula holds is shown. Given a system specification S and a LTL$_\Delta$ formula ϕ:

$S \models \phi$ iff $\forall \pi \in [[S]]_\Delta, \pi \models \phi$,
where $[[S]]_\Delta = [[S]]_w \cup \{\gamma\Delta^w \mid (\gamma, \Sigma) \in \mathit{failures}(S)\}$

Following that idea, and considering that CSP system S cannot extend its traces to consider deadlocking traces, a tester T_ϕ of the LTL formula ϕ is built. T_ϕ accepts, on the one hand, infinite traces; and on the other hand, deadlocking traces. The tester is built from a Büchi automaton using the classical approach. Therefore, given an LTL formula to check, first it is negated, second it is translated to LTL$_\Delta$, and finally it is translated to a special Büchi automaton called B_Δ. B_Δ extends the traditional Büchi automaton, adding acceptance conditions to manage deadlocks. More formally, B_Δ is defined in [9] as $\mathbf{B_\Delta} = (\mathbf{\Sigma}, \mathbf{Q}, \mathbf{T}, \mathbf{Q^0}, \mathbf{F}, \mathbf{D})$ where: Σ is the alphabet, Q is the set of states, $T \subseteq Q \times \Sigma \times Q$ is the transition relation, $Q^0 \subseteq Q$ is a set of initial states, $F \subseteq Q$ is a set of infinite trace accepting states and $D \subseteq Q$ is a set of deadlock monitor states.

B_Δ has two acceptance conditions, the classical acceptance condition for infinite traces and acceptance conditions for deadlocking traces. The traditional Büchi automaton B over the alphabet $\Sigma \cup \{\Delta\}$ is modified into B_Δ over the alphabet Σ by:

1. identifying **deadlock monitor states** (DMS) which are reachable from an initial state by transitions in Σ and accept strings Δ^w with the classical Büchi condition,
2. removing all Δ transitions,
3. removing all transitions and states which do not lead to the acceptance of a trace.

Translation from B_Δ to CSP. Each state of B_Δ is translated to a CSP process with different characteristics depending on what kind of state it is. If it is an accepting state, a CSP process with a special *success* action is created; and if it is a DMS, a special Δ transition is added. Therefore, translation from B_Δ automaton into CSP is defined as:

- map every $q \in Q$ to a CSP process name $NAME(q)$
- for every $q \in Q^0$ add the CSP definition $TESTER = NAME(q)$,
- for every non-accepting state $q \in Q \setminus F$ and
  for all outgoing edges $(q, a, q') \in T$
  add the CSP definition $NAME(q) = a \rightarrow NAME(q')$
- for every accepting state $q \in F$ where
  $\{(q, a_1, q_1), ..., (q, a_n, q_n)\} \subseteq T$ are all the outgoing edges of q
  add the CSP definition
  $NAME(q) = \mathit{success} \rightarrow (a_1 \rightarrow NAME(q_1) \, \Box \, ... \, \Box \, a_n \rightarrow NAME(q_n))$
- for every state $q \in D$
  add the CSP definition $NAME(q) = \mathit{deadlock} \rightarrow DEADLOCK$
- add a single CSP definition of $DEADLOCK$ (where $\Sigma = \{a_1, ..., a_n\}$)
  $DEADLOCK = a_1 \rightarrow k_0 \rightarrow STOP \, \Box \, ... \, \Box \, a_n \rightarrow k_0 \rightarrow STOP$

Note that $\mathit{success}$, $\mathit{deadlock}$ and k_0 are all different and not in Σ.

If the system is not deadlocked in a deadlock monitor state then the system in parallel with the $DEADLOCK$ process will be able to perform some action in Σ and then the action k_0. On the other hand, if the system is deadlocked in a deadlock monitor state, the $DEADLOCK$ process will not be able to perform k_0, so deadlocking traces will correspond to CSP failures.

Checking emptiness using FDR. The final step to verify that $[[S]]_w \cap [[\neg\phi]]_w = \emptyset$ is to check whether an infinite trace or a finite deadlocking trace of S satisfies $\neg\phi$; if there does not exist such a trace then $S \models \phi$. The way to do this check using FDR is through two kinds of refinement check.

- Traces which generate infinite successes are checked by testing whether: $SUC \sqsupseteq_T (S[[\Sigma]]TESTER) \setminus (\Sigma \cup \{deadlock, k0\})$ holds, where $SUC = success \rightarrow SUC$.
- Deadlocking acceptances are checked by testing whether: $deadlock \rightarrow STOP \sqsupseteq_F (S[[\Sigma]]TESTER) \setminus (\Sigma \cup \{success\})$ holds

If one of the tests succeeds then $S \not\models \phi$, so $S \models \phi$ only if both of them fail.

4.2 An Approach to Translate LTL Formulae from RSL to CSP

After analyzing the approach presented in [9] and [8], we take the general idea and we adapt it in order to translate LTL from RSL to FDR. In the next subsections, our framework to translate LTL properties from RSL to FDR is detailed.

LTL specification from RSL to LTL$_\triangle$. We define a grammar for writing LTL assertions in RSL. LTL properties are preceded by the key word "ltl_assertion" and the definition of each property is written using an identifier tag, the process which will be tested, and the LTL property.

LTL_prop_decl ::= "ltl_assertion" $\{$LTL_assertion$\}^+$,
LTL_assertion ::= "[" LTL_tag "]" Process_name $\vdash$ LTL_prop
LTL_prop ::= Channel_name
| LTL_prefix LTL_prop | LTL_prop LTL_infix LTL_prop
| "true" | "false" | "(" LTL_prop ")"
LTL_prefix ::= "X" | "G" | "F" | "$\sim$"
LTL_infix ::= "R" | "U" | "$\vee$" | "$\wedge$" | "$\Rightarrow$"

'Process_name' and 'Channel_name' are type identifiers defined previously for a process and a channel respectively. 'LTL_tag' is also an identifier, more precisely an LTL property identifier. For example, consider the RSL specification in Fig. 1: An ltl_assertion called "happy" is defined over the process "SYS" and the property is specified using channels "rich" and "smile". An occurrence of a channel name in an ltl_assertion indicates that the corresponding event occurs. So "happy" asserts that whenever a "rich" event occurs a "smile" will eventually occur — a classical liveness property.

The translation from RSL to LTL$_\triangle$ requires that alphabets are defined by extension, i.e. as finite sets. So LTL properties only can be defined using models involving simple channels (not channel arrays).

```
scheme VENDING_MACHINE =
 class
    channel
     rich,coin, choc, toff,smile: Unit
    value
     Machine: Unit → in coin out choc, toff Unit
     Machine() ≡ coin?;(choc!();Machine() ⊓ toff!();Machine()),

     Customer: Unit → out coin,smile in rich, choc, toff Unit
     Customer() ≡ rich?;coin!();(choc?;smile!();Customer() ⌈⌉ toff?;smile!();Customer()),

     SYS: Unit → in coin,choc,toff,smile,rich out coin,choc,toff,smile Unit
     SYS() ≡ Machine()||Customer()
 ltl_assertion
[ happy ] SYS ⊢ G(rich ⇒ F(smile))
end
```

Fig. 1. RSL specification for a simple vending machine

Negating LTL properties. We want to verify that $[[S]]_w \cap [[\neg\phi]]_w = \emptyset$ (where $[[S]]_w$ represents all the traces of the system's model and $[[\neg\phi]]_w$ all the traces of the negation of a LTL formula). So, the first step on the way to translate a LTL formula to LTL$_\Delta$, is the negation of the property by means of the introduction of the operator '$\sim$'. For this first translation we use standard rules such as $\sim$ G ϕ = F($\sim$ ϕ).

Observe that the negation of a event 'x' it is equivalent to concatenation by '$\vee$' of each alphabet's symbol, less x and plus the Δ symbol. This is because we know we have CSP events where exactly one event from the alphabet happens at any step. For the same reason, it is not possible to have more that one event concatenated by the '$\wedge$' operator either. For instance, if the alphabet is {a,b,c} then

$\sim$a = b $\vee$ c $\vee$ Δ *and* $\sim\Delta$ = a $\vee$ b $\vee$ c *and* a $\wedge$ b = false *and* a $\wedge$ Δ = false

These rules enable us to remove the negations from any formula. (We will often for clarity leave $\sim\Delta$ unexpanded in the presentation.)

Introducing the special symbol Δ. Taking as models the rules defined in [9] for translating the LTL properties $X\phi$ and $\neg X\phi$ into LTL$_\Delta$ (see subsection 4.1), we analyzed the semantics for each LTL$_\Delta$ operator. Therefore, we specify a translation T of every LTL operator of a formula ϕ as follows:

T(G(ϕ)) = G(Δ) R T(ϕ)	T($\phi \wedge \psi$) = T(ϕ) $\wedge$ T(ψ)
T(F(ϕ)) = $\sim\Delta$ U T(ϕ)	T($\phi \vee \psi$) = T(ϕ) $\vee$ T(ψ)
T(X(ϕ)) = $\sim\Delta$ $\wedge$ X(T(ϕ))	T($\phi \Rightarrow \psi$) = T(ϕ) $\Rightarrow$ T(ψ)
T(ϕ U ψ) = ($\sim\Delta$ $\wedge$ T(ϕ)) U T(ψ)	T(a) = a, where $a \in \Sigma$
T(ϕ R ψ) = (G(Δ) $\vee$ T(ϕ)) R T(ψ)	T(Δ) = Δ

Considering the VENDING_MACHINE example shown previously, the translation for the LTL assertion *happy* after the negation of the LTL property and the introduction of the symbol Δ is as follows:

T($\sim$(G(rich $\Rightarrow$ F(smile)))) = $\sim\Delta$ U (rich $\wedge$ (G(Δ) R (Δ $\vee$ rich)))

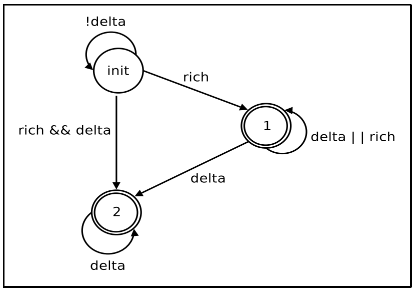


Fig. 2. Büchi automaton for the *happy* assertion

Translation from LTL$_\Delta$ to Büchi automata. The SPIN model checker [7] is used in [9] to translate LTL$_\Delta$ to Büchi automata. Instead of SPIN, we use *ltl2ba* [2] to generate the Büchi automaton from an LTL$_\Delta$ expression. We choose ltl2ba because it is open source, which allows us to extend it. In addition, experimental work shows that it is more efficient than SPIN [3].

The ltl2ba algorithm generates a Büchi automaton from an LTL formula. First a very weak alternating automaton is built and then it is transformed it into a Büchi automaton, using a generalized Büchi automaton. Each automaton is simplified on-the-fly for saving memory and time, using iteratively three rules until no more simplification are possible [3]:

1. Inaccessible states are removed.
2. If a transition t_1 implies a transition t_2, then t_2 is removed.
3. If states q_1 and q_2 are equivalent, then they are merged.

The Büchi automaton for the *happy* assertion is shown in figure 2.

Translation from Büchi automata to Büchi delta B_Δ. We took the source code of *ltl2ba* and extended it to generate B_Δ from the Büchi automaton. B_Δ is obtained following the steps shown in subsection 4.1, that is: 1) identifying each DMS, 2) removing all Δ transitions and finally 3) removing transitions and states which do not lead to the acceptance of a trace.

Regarding detection of DMS, in [9] they are found following an algorithm defined in [17] (adaptation of the Tarjan's search algorithm for strongly connected components). However, this one only detects states which accept strings Δ^w with the classical Büchi condition, but it does not consider which are reachable from an initial state by transitions in Σ. Therefore, we defined a complementary algorithm which detects states reachable by Σ transitions.

Consider the Büchi automaton corresponding to the VENDING_MACHINE example. Only state 1 is detected as DMS, because it is reachable from the initial state by transitions in Σ and accepts the string Δ^w with the classical Büchi condition. Also, Δ transitions are removed, according to step 2) for building B_Δ. The transition labelled 'rich && delta' is removed too, because we know we have CSP events where exactly one event from the alphabet happens at any time. Therefore, we get the B_Δ automaton shown in figure 3.

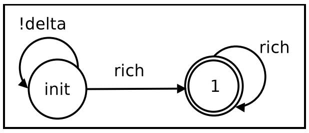


Fig. 3. B_Δ automaton for the *happy* assertion

Translation from B_Δ to CSP and emptiness check. After building B_Δ, we use the algorithm of section 4.1 for generating the corresponding CSP specification from B_Δ. Also, we generate FDR code for the two refinement checks as explained in subsection 4.1; i.e. checking traces which generate infinite successes and checking for deadlock.

5 Implementation of the Tool

The approach described in this paper has been implemented in a tool which translates RSL specifications with temporal logic assertions into a CSP_M specification and a tester process.

In order to do so it is set up with two main components that we will call "RSL_FDR2" and "RSL_LTL_FDR2".

RSL_FDR2 takes an RSL specification as an input and performs two things. First, the RSL is transformed into an AST (abstract syntax tree) by the RSL type checker; then, applying many translation rules, the AST of RSL is transformed into an AST of CSP_M; and finally this AST is converted into a new output script (a `.fdr2` file) in CSP_M. Second, if one or more LTL assertions are specified (using RSL syntax), RSL_FDR2 translates them to the corresponding LTL$_\Delta$ formulae and saves them in `.ltl` files. These `.ltl` files are input to RSL_LTL_FDR2, an extension of ltl2ba, which generates for each a `TESTER` process and some CSP refinement statements, and appends them to the `.fdr2` file.

For instance, when we give the VENDING_MACHINE example of Section 4 to the tool, it produces the following CSP_M script as output:

```
channel rich,coin,choc,toff,smile

Alph_in_Machine = {|coin|}
Alph_out_Machine = {|toff,choc|}
Machine = coin -> (choc -> (Machine) [] toff -> (Machine))

Alph_in_Customer = {|toff,choc,rich|}
Alph_out_Customer = {|smile,coin|}
Customer = rich -> (coin -> (choc -> (smile -> (Customer)) |~|
                             toff -> (smile -> (Customer))))

Alph_in_SYS = {|rich|}
Alph_out_SYS = {|smile|}
SYS = (Machine [{|coin,toff,choc|}||
                {|toff,choc,rich,smile,coin|}] Customer)\
                                            {|coin,toff,choc|}
```

```
channel success0,deadlock0,k0
Alph_SYS0 = union(Alph_in_SYS,Alph_out_SYS)

TESTER0 = State0_0
State0_0 =
 rich?x -> State1_0 [] rich?x -> State0_0 [] smile?x -> State0_0
State1_0 =
  success0 -> ( rich?x -> State1_0 ) [] deadlock0 -> DEADLOCK0
DEADLOCK0 = rich?x -> k0 -> STOP [] smile?x -> k0 -> STOP
Composition0 =
  (SYS [|Alph_SYS0|] TESTER0)\ union(Alph_SYS0,{deadlock0,k0})
DComposition0 =
  (SYS [|Alph_SYS0|] TESTER0)\ union(Alph_SYS0,{success0})

SUC0 = success0 -> SUC0
assert Composition0 [T= SUC0
RealDeadlock0 = deadlock0 -> STOP
assert DComposition0 [F= RealDeadlock0
```

The Composition0 and DComposition0 assertions are checked using FDR to determine if the property holds. Compostion0 checks for traces which generate infinite successes and Dcompositon0 checks for deadlock. Since both assertions fail, it is established that SYS satisfies the property G(rich $\Rightarrow$ F(smile)).

5.1 Efficiency

As suggested by the fact that there is a bisimulation between the RSL description and its translation, there is a one-to-one relation between the events in RSL and those in CSP, and the translation is almost certainly as good and as efficient in model checking as a hand translation would be. This has been borne out by trying the translator on a number of standard examples: cyclic scheduler, dining philosophers, railway crossing, producer-consumer, alternating bit protocol, multiplexed buffer [18].

6 Conclusions

We have shown an approach to translate a concurrent applicative subset of RSL into CSP_M, and shown the soundness of the translation through establishing a strong bisimulation. We have analyzed the approach presented in [9] and [8] and we have observed it is possible to take the general idea but it is necessary to adapt it in order to translate from RSL to CSP. Therefore, we have shown the whole translation of every LTL operator to LTL$_\Delta$ operators, we have used *ltl2ba* algorithm to translate from LTL expressions to Büchi automata and we have shown how to extend *ltl2ba* to build B$_\Delta$.

Finally, we have developed a tool for the specification of concurrent systems that allows us, first, to use the FDR tool on the CSP_M scripts, and to draw sound conclusions about the RSL descriptions and second, to translate LTL formulas from RSL to CSP that helps us to express the specification of desired properties in a friendly way enabling the model checking of LTL formulae about RSL descriptions with FDR.

A problem with this approach is that it needs failure of model checking to prove success. So if the proposed property is not proved, there is only model checking success and no trace to indicate what went wrong. By including a notion of fairness in the model checking it may be possible to prove LTL properties more directly, as hinted by Roscoe [15] and adopted in recent work in Singapore [11].

References

1. Debabi, M.: The RSL semantic course (1993)
2. Gastin, P., Oddoux, D.: ltl2ba tool, http://www.lsv.ens-cachan.fr/~gastin/ltl2ba/index.php
3. Gastin, P., Oddoux, D.: Fast LTL to büchi automata translation. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 53–65. Springer, Heidelberg (2001)
4. George, C.: RAISE Tools User Guide. Technical Report 227, UNU-IIST, P.O. Box 3058, Macau (February 2001), http://www.iist.unu.edu
5. The RAISE Language Group. The RAISE Specification Language. Prentice-Hall, Englewood Cliffs (1992)
6. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
7. Holzmann, G.J.: The SPIN Model Checker. Addison-Wesley, Reading (2003)
8. Leuschel, M., Massart, T., Currie, A.: How to Make FDR Spin LTL Model Checking of CSP by Refinement. Journal Lecture Notes in Computer Science 2021, 99+ (2001)
9. Leuschel, M., Massart, T., Currie, A.: How to Make FDR Spin LTL Model Checking of CSP by Refinement, Technical Report, Dependable Systems and Software Engineering Research Group, School of Electronics and Computer Science, University of Southampton, England (September 2000)
10. Formal Systems (Europe) Ltd. Failures-Divergence Refinement: FDR2 User Manual (2005)
11. PAT: Process Analysis Toolkit, http://www.comp.nus.edu.sg/~pat/
12. Pnueli, A.: The temporal logic of concurrent programs. Theoretical Computer Science 13, 45–60 (1981)
13. The RAISE Method Group. The RAISE Development Method. Prentice-Hall International, Englewood Cliffs (1995)
14. Roscoe, A.W.: Model-checking CSP, pp. 353–378. Prentice Hall International (UK) Ltd., Hertfordshire (1994)
15. Roscoe, A.W.: Compiling Shared Variable Programs into CSP. In: Proceedings of PROGRESS workshop 2001 (2001)
16. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall, Englewood Cliffs (2005)
17. Sedgewick, R.: Algorithms in C++. Addison-Wesley, Reading (1992)
18. Tapia, L., George, C.: Model Checking Concurrent RSL with CSPM and FDR2. Research Report 393, UNU-IIST, P.O.Box 3058, Macau (April 2008)
19. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Proccedings of LIC6, pp. 332–344 (1986)
20. Vargas, A.P., George, C.: Formalising the translation from RSL to CSP. Research Report 395, UNU-IIST, P.O.Box 3058, Macau (May 2008)

An Introduction to Grammar Convergence

Ralf Lämmel and Vadim Zaytsev

Software Languages Team, The University of Koblenz-Landau, Germany

Abstract. Grammar convergence is a lightweight verification method for establishing and maintaining the correspondence between grammar knowledge ingrained in all kinds of software artifacts, e.g., object models, XML schemas, parser descriptions, or language documents. The central idea is to extract grammars from diverse software artifacts, and to transform the grammars until they become syntactically identical. The present paper introduces and illustrates the basics of grammar convergence.

1 Introduction

Grammar convergence is a lightweight verification method for establishing and maintaining the correspondence between grammar knowledge ingrained in all kinds of software artifacts. In fact, it is an integrated method that works purposely across different programming and specification languages as well as different approaches to software development. Here are few use cases for grammar convergence:

– Given are Java classes for a specific domain, say financial exchange. There is also an independently designed XML schema that is meant to standardize that domain. One needs to establish the agreement between the object model and the schema.
– Given is a compiler for a programming language, say gcc for C++. There is also a reverse/re- engineering tool for the same language based on a different parsing infrastructure. One needs to establish that both tools agree on the language at hand.
– Given is an XML-data binding technology. One needs to test the (customizable) mapping from XML schemas to object models. The oracle for testing relies on establishing an agreement between XML schemas and object models.
– Given are 3 versions of the Java language specification, with 2 grammars per version. One needs to align grammars per version and express the evolution from version to version. (We have done such a case study; see the authors' website.)

The central idea of grammar convergence is to extract grammars from diverse software artifacts, and to transform the grammars until they become syntactically identical. In more detail, the method entails the following core ingredients:

1. A unified *grammar format* that effectively supports abstraction from specialities or idiosyncrasies of the grammars as they occur in software artifacts in practice.
2. A *grammar extractor* for each kind of artifact – e.g., a Java extractor maps Java classes to the unified grammar format.
3. A *grammar comparer* that determines and reports grammar differences in the sense of deviations from syntactical equality (if any).

4. A framework for automated *grammar transformation* that can be used to refactor, or to otherwise more liberally edit grammars until they become syntactically identical.

The method also entails the following optional ingredients:

5. Grammar convergence may be extended to the 'instance level' so that instances (such as parse trees or XML documents) are also extracted, compared and transformed.
6. The transformations of grammar convergence may be semi-automatically derived ('inferred') from grammar differences.

The present paper only covers the core ingredients 1.-4.

Contributions

– Grammar convergence helps in relating grammar knowledge that is readily ingrained in diverse forms of software artifacts; it complements the use of generative (or model-driven) approaches, when they are not used, have not been used, or cannot (yet) be used.
– Grammar convergence delivers conceptually simple grammar transformations to software artifacts of kinds that would normally require more complicated transformations, e.g., XML schemas, and object models. This is possible because of the abstraction done during extraction.
– An implementation of grammar convergence is publicly available.[1]

Roadmap. §2 describes the basics of grammar convergence and introduces the running example of the paper. §3 outlines BGF — the BNF-like Grammar Format, i.e., the unified grammar format that we use in our implementation of grammar convergence. §4 describes and illustrates the concept of grammar extraction. §5 sketches our suite of programmable transformations for grammar convergence. §6 discusses related work. §7 concludes the paper.

2 Basics and Running Example

We use a trivial programming language FL ('Factorial Language'; available from the quoted repository) as a running example. That is, we converge grammars for FL that were obtained from different FL language processors such as interpreters and optimizers. Here is an illustrative program in the FL language; it defines two functions: one for multiplication; another for the factorial function; the latter in terms of the former:

```
mult n m = if (n == 0) then 0 else (m + (mult (n − 1) m))
fac n = if (n == 0) then 1 else (mult n (fac (n − 1)))
```

2.1 Sources of Convergence

Fig. 1 shows a convergence tree for some FL components. The *leafs* of the tree (see at the top) denote different *sources*. We use the term source to mean 'software artifact containing grammar knowledge'. Here is short description of the sources for FL:

[1] https://sourceforge.net/projects/slps/

antlr. This is a parser description in the input language of ANTLR[2]. Semantic actions (in Java) are intertwined with EBNF-like productions.
dcg. This is a logic program written in the style of definite clause grammars; c.f. Fig. 2.
sdf. This is a concrete syntax definition in the notation of SDF/SGLR[3] with scannerless generalized LR parsing as parsing model; c.f. Fig. 3.
xsd. This is an XML schema[4] for the abstract syntax of FL.
om. This is a hand-crafted object model (Java classes) for the abstract syntax of FL.
jaxb. This object model was generated by JAXB[5] from an XML schema for FL.

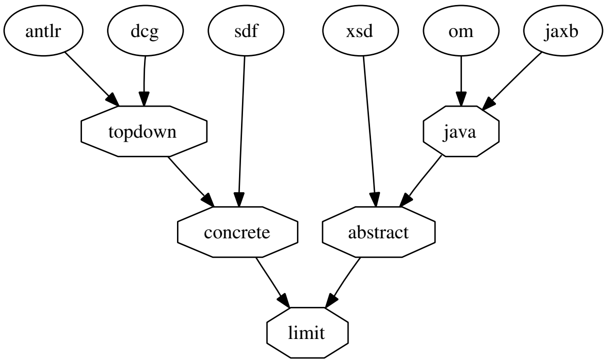


Fig. 1. The overall convergence graph for the 'Factorial Language'

```
program(Fs)  --> +(function,Fs).
function (N,Ns,E)) --> name(N), +(name,Ns), @("="), expr(E), +(newline).

expr(E)                          --> lassoc(ops,atom,binary,E).
expr(apply(N,Es))                --> name(N), +(atom,Es).
expr(ifThenElse(E1,E2,E3))       --> reserved("if"), expr(E1), ...

atom( literal (I))      --> int(I).
atom(argument(N))       --> name(N).
atom(E)                 --> @("("), expr(E), @(")").

ops(equal)   --> @("==").
ops(plus)    --> @("+").
ops(minus)   --> @("−").
```

Fig. 2. Definite Clause Grammar for FL (The clauses construct a term representation; see the arguments of the various predicates. The DCG leverages higher-order predicates for EBNF-like expressiveness and left-associative tree construction (c.f., '+' and 'lassoc'). The priorities on expression forms are expressed by means of a layered definition; c.f., 'expr' vs. 'atom'.)

[2] `http://antlr.org`
[3] `http://www.program-transformation.org/Sdf/SGLR`
[4] `http://www.w3.org/XML/Schema`
[5] `http://jaxb.dev.java.net/`

Function+	→ Program	
Name Name+ ”=” Expr Newline+	→ Function	
Expr Ops Expr	→ Expr	{**left**,**prefer**,**cons**(binary)}
Name Expr+	→ Expr	{**avoid**,**cons**(apply)}
”if” Expr *”then”* Expr *”else”* Expr	→ Expr	{**cons**(ifThenElse)}
”(” Expr *”)”*	→ Expr	{**bracket**}
Name	→ Expr	{**cons**(argument)}
Int	→ Expr	{**cons**(literal)}
”−”	→ Ops	{**cons**(minus)}
”+”	→ Ops	{**cons**(plus)}
”==”	→ Ops	{**cons**(equal)}

Fig. 3. SDF grammar for FL (Only (context-free) SDF productions are shown. Notice that the ‘defining expression’ of a production appears on the left side of the arrow, and the ‘defined nonterminal’ on the right side. Productions can be annotated in certain ways between the braces, e.g., with constructor names (c.f., cons), or directions for disambiguation (c.f., prefer, avoid).)

2.2 Targets of Convergence

Consider again Fig. 1. The *inner nodes and the root* denote a number of *targets* for FL. We use the term target to mean ‘derived grammars that establish the correspondence between some sources’. Here is short description of the targets for FL:

topdown. The sources *antlr* and *dcg* both leverage top-down parsing. Their correspondence can be established by a few simple refactoring steps.

concrete. This target converges all concrete syntax definitions. A noteworthy difference is that *sdf* uses one expression nonterminal, whereas *topdown* uses two ‘layers’.

java. The sources *om* and *jaxb* are both object models whose correspondence can be established by simple refactoring steps.

abstract. The target *java* and the XML schema are to be converged to an abstract syntax definition. The corresponding refactorings need to neutralize the style differences implied by the data models: OO vs. XML.

limit. The targets *concrete* and *abstract* are converged (to an even more abstract syntax). For instance, terminals are removed from *concrete*.

3 BGF — BNF-Like Grammar Format

3.1 Design Rationale

In principle, we could try to leverage an existing syntax definition formalism (e.g., SDF/SGLR (see an earlier footnote) or a meta-modeling facility (e.g., EMF[6]). In contrast, we have derived BGF such that it covers the grammar-like expressiveness that we encountered in different kinds of software artifacts. Also, BGF allows us to avoid any sort of bias towards a particular parsing model or other details of operational semantics. For convenience, we can still represent BGF in other notations (using a generative approach as in [13]).

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

3.2 BGF Concepts

We start with the most trivial aspects:

- Terminals and nonterminals.
- Regular expression-like composition and grouping:
  - Sequential composition (infix ‘,’ – also called ‘sequence’).
  - Alternative composition (infix ‘;’ – also called ‘choice’).
  - Epsilon (’true’) and the ‘empty language’.
  - Iteration and optionality (‘*’, ‘+’, ‘?’).
- A production is a pair of ‘defined nonterminal’ and ‘defining expression’.
- A grammar consists of a set of start symbols and a set of productions.

At this point, we have reached ‘representation capability’ for textbook-style BNF and EBNF (when restricted to context-free syntax). Only few more concepts are needed to represent essential extras of XML schemas, object models, and algebraic signatures:

Production labels. Extraction may populate these labels from names of OO subclasses, derived XML schema types, or algebraic term constructors. As a bonus, labels are convenient in addressing productions in programmable grammar transformations.

Expression selectors. While a flat record-like grammar structure with top-level selectors is sufficient to represent typical object models, more liberal selectors are needed to represent arbitrarily nested element declarations of XML Schema.

Simple types. Types such as string and int are added to cover the simple types used in algebraic data types, object models, and XML schemas. Even syntax definitions indirectly involve simple types through the attributes associated with lexemes.

Universal type. This type is a fallback for extraction whenever no precise grammar structure can be determined, e.g., when mapping the OO base type ‘object’, wildcards of XML Schema, or dynamics in functional programming.

Namespaces. Various kinds of sources are organized in namespaces, c.f., Java’s packages for object models, Haskell’s hierarchical module system, or XML Schema’s foundation on XML namespaces. Such organization can be preserved by extraction.

3.3 Self-representation

In our implementation, BGF is primarily defined by an XML Schema, which naturally, is too voluminous to be shown here. While the XML-based representation of BGF grammar may be convenient for data exchange, several of our components use a Prolog-based term notation. (For instance, the transformation component is implemented in Prolog.) Fig. 4 lists the BGF of BGF in the Prolog-based term notation.[7] The interesting status of the shown grammar is that it has been computed from the BGF that was extracted from the primary XML schema for BGF. That is, we have applied grammar convergence to align the XML schema with the expected Prolog-based term notation for BGF.

[7] The notation uses predominantly prefix terms with the exception of special list notation and infix functors ‘,’ and ‘;’ for sequences and choices. Note that optionality is represented via lists — using the empty list [] for the case of absence, and the singleton list otherwise. Certain symbols need to be escaped by quotes or parentheses, as one can see in the figure.

```
g( [g,p,x,v,l,n,s,t], [
  p([g], g, (*(n(n)), *(n(p)))),         -- grammar = start symbols + productions
  p([p], p, (?(n(l)), n(n), n(x))),      -- production = label + LHS + RHS
  p([true], x, true),                   -- epsilon
  p([ fail ], x, true),                 -- empty language
  p([v], x, n(v)),                      -- values of simple types
  p([a], x, true),                      -- all ( universal type)
  p([t], x, n(t)),                      -- terminals
  p([n], x, n(n)),                      -- nonterminals
  p([s], x, (n(s), n(x))),              -- selector expressions
  p([(',')], x, (n(x), n(x))),          -- sequence
  p ([(;)], x, (n(x), n(x))),           -- choice
  p ([?], x, n(x)),                     -- optionality
  p ([+], x, n(x)),                     -- 1 or more repetitions
  p ([*], x, n(x)),                     -- 0,1 or more repetitions
  p([int], v, true),                  -- integer values
  p([string], v, true),               -- strings
  p ([], l, v(string)),             -- labels are strings
  p ([], n, v(string)),             -- nonterminal symbols are strings
  p ([], s, v(string)),             -- selectors are strings
  p ([], t, v(string))              -- terminal symbols are strings
])
```

Fig. 4. BGF of BGF w/o namespaces

4 Grammar Extraction

4.1 Abstraction by Extraction

The limited expressiveness of BGF, when compared to any possible source format, implies that some of the details of the source format are not conveyed into the extracted grammar; we call this effect 'abstraction by extraction'. Such abstraction simplifies proofs of grammar correspondences at the cost of potentially missing certain kinds of grammar differences. Here are examples of details that are abstracted away in this manner; they are grouped by kinds of grammarware:

Parser descriptions
- Semantic actions
- Lexical syntax descriptions
- Precedence declarations

Object models
- Constructors, static methods, initializers
- Specific types of collection classes
- Distinction of classes vs. interfaces and fields vs. methods

Algebraic data types
- Distinction of nominal types vs. types aliases
- Higher-order and quantified types (represented universally)

BGF extracted from the SDF for FL as of Fig. 3

```
p([binary], 'Expr', (n('Expr'), n('Ops'), n('Expr'))),
p([apply], 'Expr', (n('Name'), +n('Expr'))),
p([ifThenElse], 'Expr', (t(if), n('Expr'), t(then), n('Expr'), t(else), n('Expr'))),
p([], 'Expr', (t('('), n('Expr'), t(')'))),
p([argument], 'Expr', n('Name')),
p([literal], 'Expr', n('Int')),
p([minus], 'Ops', t(−)),
p([plus], 'Ops', t(+)),
p([equal], 'Ops', t(==))
```

BGF extracted from the DCG for FL as of Fig. 2

```
p([binary], expr, (n(atom), *((n(ops), n(atom))))),
p([apply], expr, (n(name), +n(atom))),
p([ifThenElse], expr, (t(if), n(expr), t(then), n(expr), t(else), n(expr))),
p([literal], atom, n(int)),
p([argument], atom, n(name)),
p([], atom, (t('('), n(expr), t(')'))),
p([equal], ops, t(==)),
p([plus], ops, t(+)),
p([minus], ops, t(−))
```

BGF extracted from an ANTLR frontend for FL

```
p([], expr, (n(binary);n(apply);n(ifThenElse))),
p([], binary, (n(atom), *((n(ops), n(atom))))),
p([], apply, (n('ID'), +n(atom))),
p([], ifThenElse, (t(if), n(expr), t(then), n(expr), t(else), n(expr))),
p([], atom, (n('ID');n('INT');t('('), n(expr), t(')'))),
p([], ops, (t(==);t(+);t(−)))
```

BGF extracted from an XML schema for FL

```
p([], 'Function', (s(name, v(string)), +s(arg, v(string)), s(rhs, n('Expr')))),
p([], 'Expr', (n('Literal');n('Argument');n('Binary');n('IfThenElse');n('Apply'))),
p([], 'Literal', s(info, v(int))),
p([], 'Argument', s(name, v(string))),
p([], 'Binary', (s(ops, n('Ops')), s(left, n('Expr')), s(right, n('Expr')))),
p([], 'Ops', (s('Equal', true);s('Plus', true);s('Minus', true))),
p([], 'IfThenElse', (s(ifExpr, n('Expr')), s(thenExpr, n('Expr')), s(elseExpr, n('Expr')))),
p([], 'Apply', (s(name, v(string)), +s(arg, n('Expr'))))
```

Fig. 5. Some extraction results for FL (Only expression syntax is shown.)

XML schemas

- Distinction of elements, attributes, complex types, and groups
- Simple type constraints

4.2 Grammar Extractors

An extractor is simply a software component that processes a software artifact and produces a (BGF) grammar. In the typical case, extraction boils down to a straightforward mapping defined by a single pass over the input. Extractors are preferably implemented within the computational framework of the source artifact at hand, or in its affinity, e.g.:

- ANTLR: ANTLR
- DCG: Prolog

– Java: java.lang.reflect or com.sun.source.tree
– SDF: ASF+SDF Meta-Environment[8] or Stratego/XT[9]

On the output side, an extractor leverages the XML format for BGF.

4.3 Extraction Samples

Fig. 5 contrasts the extraction results for several FL sources. The differences between the grammars can be summarized as follows:

– Only the ANTLR&DCG&SDF extracts contain terminals.
– The ANTLR&DCG extracts contain expressions layers expr and atom.
– The SDF&XSD extracts contain a single expression layer.
– Only the XSD extract contains selectors.
– The ANTLR&XSD extracts leverage choices.
– The DCG&SDF extracts leverage nonterminals with multiple productions.
– There is also some variation on using production labels.
– More trivially, the grammars disagree on names, upper and lower case.

5 Programmable Grammar Transformations

In the more preferable case, two different grammars can be refactored to become syntactically identical. We use the term *refactoring* in the established sense of semantics-preserving transformations. In the less preferable case, non-semantics-preserving transformations are due, in which case weaker properties should limit the impact.

5.1 Transformation Properties

We may refer to the semantics of a grammar as the language (set of strings) generated by the grammar, as it is common for formal languages — for context-free grammars, in particular. With the string-oriented semantics in mind, few transformations are semantics-preserving. Examples include renaming of nonterminals, and fold/unfold manipulations. To give an example where different semantics are needed consider the scenario of aligning a concrete and an abstract syntax.

When necessary, we may apply the algebraic interpretation of a grammar, where grammar productions constitute an algebraic signature subject to a term-algebraic model. In this case, the terminal occurrences in any given production do no longer carry semantic meaning; they are part of the function symbol. (Hence, abstract and concrete syntaxes can be aligned now.) Some transformations that were effortlessly semantics-preserving w.r.t. the string-oriented semantics, require designated bijective mappings w.r.t. the term-oriented semantics, e.g., fold/unfold manipulations, but generally, the term-oriented semantics admits a larger class of semantics-preserving transformations than the string-oriented one.

[8] http://www.meta-environment.org/
[9] http://strategoxt.org/

For brevity, we omit the discussion of another alternative: graph-oriented semantics.

Transformations that are not semantics-preserving may still be 'reasonable' if they model data refinement [8,25].[10] A simple way to think of data refinement in our context is that a transformation increases or decreases the number of 'representational options', e.g., by making a certain syntactic structure optional or mandatory. Here we assume the term-oriented semantics with its term-algebraically defined domains.

Some grammar differences may require more arbitrary replacements. In this case, one would want to be sure that *a)* indeed no more preserving transformation is possible, and *b)* the scope of replacement is as small as possible. To this end, we have developed an effective strategy, which however is beyond the scope of the present paper.

5.2 Grammar Refactoring

Let us demonstrate a number of refactoring operators. In our running example, there are two sources that are very close to each other: *antlr* and *dcg*; c.f., Fig. 5. Both sources serve top-down parsing. The remaining differences are neutralized by the following refactorings to be applied to the ANTLR grammar; we show the applications of the transformation operators combined with an explanatory comment:

```
renameN('NEWLINE', newline)                      % use lower case
renameN('INT', int)                              % use lower case
renameN('ID', name)                              % rename ID to name
verticalN(expr)                                  % many expr productions
unchain(p([], expr, n(apply)))                   % inline apply production
unchain(p([], expr, n(binary)))                  % inline binary production
unchain(p([], expr, n(ifThenElse)))              % inline ifThenElse
verticalN(atom)                                  % many atom productions
deanonymize(p([literal], atom, n(int)))          % add label for literals
deanonymize(p([argument], atom, n(name)))        % add label for arg refs
verticalN(ops)                                   % many ops productions
deanonymize(p([equal], ops, t(==)))              % label == production with equal
deanonymize(p([plus], ops, t(+)))                % label + production with plus
deanonymize(p([minus], ops, t(−)))               % label − production with minus
```

Fig. 6 briefly describes a small suite of refactoring operators. All operators except ***permute*** are semantics-preserving w.r.t. string-oriented semantics. Without exception, the operators are semantics-preserving w.r.t. term-oriented semantics.

5.3 Grammar Editing

We use the term *grammar editing* for transformations that go beyond refactoring. Let us consider an example. The *antlr* and *dcg* sources of FL use two expression layers (expr and atom), whereas the *sdf* source only uses one expression layer (and deals with

[10] We say that a data type (domain) A can be refined to a data type (domain) B, denoted by the inequality $A \leq B$, if there is an injective, total function $to : A \to B$ (the representation function), and a surjective, possibly partial function $from : B \to A$ (the abstraction function) such that $from.to = id_A$, where id_A is the identity function on A.

renameN(N_1, N_2) renames all occurrences of the nonterminal N_1 to N_2, provided N_2 does not occur in G. There are also operators ***renameL*** and ***renameS*** for renaming labels and selectors. In ***renameS***(OL, S_1, S_2), OL is an optional label; if present, S_1 is renamed only in the scope of the identified production, or globally otherwise.

permute(P) replaces a production say P' by P, where P and P' must agree on their defined nonterminal and (optional) label while their defining expressions must be permutations of each other (with regard to sequential composition). Here is an example:
- A production: **p**([binary], expr, (**n**(expr), **n**(ops), **n**(expr)))
- A permutation: **p**([binary], expr, (**n**(ops), **n**(expr), **n**(expr)))

verticalN(N) converts the choice-based definition of N to multiple productions. Each alternative of the choice becomes another production. An outermost selector, if present, is reused as a production label (but must not yet be in use in G). The variation ***verticalP***(P) limits the conversion to a production P. There is the opposite operator ***horizontal***.

unchain(P) replaces a chain production P and the production P' that defines the nonterminal of its defining expression by a production that inlines P' in P. (There is also the opposite operator ***chain***.) Here is an example:
- The chain production: **p** ([], expr,**n**(literal))
- The referenced definition: **p** ([], literal ,**n**(**int**))
- The result of unchaining: **p**([literal], expr,**n**(**int**))

deanonymize(P) replaces an unlabeled production say P' by its labeled variant P. There is also the opposite operator ***anonymize***.

lassoc(P) replaces list-based recursion by binary recursion. (The 'l' in ***lassoc*** is for *left* association hinting at the expected effect at the instance level. There is also an operator ***rassoc*** hence.) Here, P describes binary recursion. Their must be a corresponding production in G that uses list-based recursion. Here is an example:
- Binary recursion: **p**([binary], expr, (**n**(expr), **n**(ops), **n**(expr)))
- List-based recursion: **p**([binary], expr, (**n**(expr), $*$((**n**(ops), **n**(expr)))))

Fig. 6. Operators for grammar refactoring (G refers to the input grammar.)

priorities by extra annotations). The following transformation uses an editing operator ***unite*** to merge the two layers (i.e., nonterminals) in one:

unite (atom, expr)

Consider another example. The grammars in Fig. 5 differ with regard to the grammatical details regarding FL's literals and function or argument names. The *xsd* source uses precise (simple) types int and string, whereas the other grammars leave the corresponding nonterminals undefined (because the extraction only returned immediate context-free structure in those cases). The following transformations resolve the undefined nonterminals in accordance to the *xsd* source:

```
define ([p ([], name, v( string ))])    % names are strings
define ([p ([], int , v( int ))])       % ints ( literals ) are ints
```

Consider a final example. The convergence of concrete and abstract syntax definitions requires a transformation that removes all details that are specific to concrete syntax definitions. That is, we project away the reference to newline, strip off all terminals,

project(P) replaces a production say P' by P, where P and P' must agree on their defined nonterminal and (optional) label, and the defining expression of P must be a sub-sequence of the one of P' (with regard to sequential composition).

stripTs removes all terminals.

stripSs removes all selectors.

skip(P) removes a reflexive chain production P.

unite(N_1, N_2) recursively merges the definitions of N_1 and N_2 into one by replacing all defining and using occurrences of N_1 by N_2.

define(Ps) adds the productions Ps as a definition, assuming that all productions agree on a defined nonterminal that is used but not yet defined in G. We take the view that an undefined nonterminal is implicitly defined to be equal to the universal type. Hence, the ***define*** operator essentially 'narrows' a definition in a semantic sense. There is also the opposite operator ***undefine*** for discarding the explicit definition of a nonterminal.

Fig. 7. Operators for grammar editing (G refers to the input grammar.)

remove the bracketing production, and permute the ingredients of binary expressions to resemble prefix instead of infix notation. Thus:

```
project (p ([], function , (v( string ), +v( string ), t (=), n(expr ))))
stripTs
skip (p ([], expr, n(expr )))
permute(p([binary ], expr, ’, ’([n(ops ), n(expr ), n(expr )])))
```

Fig. 7 briefly describes a small suite of editing operators. In fact, the editing operators ***stripTs*** and ***stripSs*** are semantics-preserving w.r.t. the term-oriented semantics because terminals and selectors are irrelevant for interpreting a grammar as a signature. All but one of the remaining operators model data refinement in one direction or the other, i.e., from input (I) to output (O), or vice versa: ***skip***: $O \leq I$, ***unite***: $I \leq O$, ***define***: $O \leq I$, ***undefine***: $I \leq O$. The operator ***project*** does not model data refinement; rather it models 'data disposal'. Its I-to-O mapping for ***project*** is total, surjective, non-injective; its O-to-I mapping is not generally defined.

6 Related Work

Interoperability. The consistent use of structural and nominal types (to be compared here with grammar knowledge) is a goal shared with programming-language type systems, exchange formats, and interface definition languages (IDLs). IDLs are specifically used in distributed programming. Exchange formats are widely used for any sort of data- and communication-intensive programming. A domain with classic grammar-like exchange formats including bridges between different formats is reverse engineering [9, 14]. In the broad context of interoperability, *grammar convergence provides added value in the situation where diverse, related grammar-like knowledge is ingrained in different software artifacts.* The use of extraction and transformation compensates for the lack of consistent use of a common type system, IDL, or exchange format, and it allows for flexible correspondence relationships.

Testing grammarware. The I/O behavior of grammarware (e.g., the acceptor behavior of a frontend) can be tested by 'sampling' — subject to test-data generation and test suites [16, 19, 23, 31]. Such approaches are specifically useful for differential testing of grammarware. *Grammar convergence is complementary in that it provides a static verification of the correspondence between different software artifacts based on access to the internal structure of the artifacts.* It can also be applied to specify the 'distance' between grammars.

Generators and synchronizers. If two artifacts are meant to use the same grammar (type, etc.) modulo its realization in the software artifact, then, arguably one grammar (or software artifact) should be generated from the other. One scenario of that kind is XML-object mapping where object models are derived from XML schemas or vice versa [18]. Another scenario is the provision of text-to-model and model-to-text capabilities in model-driven engineering, where, for example, a parser description may be generated from a sufficiently rich ('annotated') metamodel [11].

One may go beyond generation, and even require bidirectional synchronization between scattered grammar knowledge, akin to bidirectional model/model or model/code synchronization in model-driven engineering [32]. As should be clear from the list of use cases in the introduction, *grammar convergence is applicable even when generators or bidirectional synchronizers are not, have not been, or cannot (yet) be used for whatever technical or other reason*. In particular, existing components do not need to be adapted, in any way, when applying the method of grammar convergence.

As an illustration, let us consider two concrete scenarios. First, consider the problem of different versions of a highly idiosyncratic parser description [27]. Bidirectional synchronization is not available in this context, but grammar convergence applies, and establishes the correspondence between the grammar versions. Second, consider the derivation of a technology-specific parser description from a technology-neutral baseline grammar. Only simpler cases of this process can be automated [13, 11].

Grammar recovery. Our work is heavily influenced by the idea of grammar recovery [2, 6, 10, 17, 20, 30], especially those forms that begin with the extraction of grammar knowledge from an artifact like a standard (containing syntax) or an implementation (based on an idiosyncratic parsing technology). Just like grammar convergence, grammar recovery involves (manual or automated) grammar transformations, which we discuss below. While grammar recovery has focused on (mostly concrete) syntax definitions, grammar convergence applies to a very broad interpretation of grammars (XML schemas, object models, etc.). Grammar recovery is a reverse-engineering method that relies on conservative parser testing to derive a quality grammar from the source. In contrast, *grammar convergence is a verification method that establishes and maintains grammatical correspondences between software artifacts*.

Grammar transformation. (Automated) grammar transformation has seen a surge of interest over the last decade, but the concept is much older because parsing technologies tend to require some internal transformations, c.f., the classic example of left-recursion removal [1, 22, 24]. There are several modern use cases for grammar transformation that support automated software engineering and grammar-based programming in one way or another: grammar recovery (see above), derivation of an abstract from a concrete syntax [33], problem-specific customization of grammars [4], and mediation between different grammar classes [28]. Ultimately, we speak of grammar programming or programmable grammar transformations [3].

Grammar convergence relies on an advanced operator suite for grammar transformation the design of which is driven by the unified grammar format, and the kinds of grammar differences that we have encountered. The design of the operator suite has not yet fully stabilized; we are still pursuing research on principled properties of grammar transformations — at all levels: single operators, operator suites, and sequences of operator applications. This work is based on earlier research by the first author [15,21].

Grammar convergence. Finally, we mention grammar engineering techniques that can be seen as specific forms of grammar convergence. In [2], the compatibility of (different implementations of) precedence rules in grammars is checked. Our (current) grammar convergence approach does not address any parsing techniques specifically, but in return, it is more generic (with regard to the notion of grammar), and programmable (with regard to deltas between grammars). In [12], the correspondence between a concrete and abstract syntax definition is addressed: the specifications for both syntaxes may be incomplete, as long as they complement each other consistently. Grammar convergence provides a general tool for 'programming' such relationships, and verifying them. In [27], the problem of proliferation of grammar-based artifacts (in fact, parser descriptions with semantic actions) due to grammar evolution or always new grammar use cases is addressed. Based on ideas of version control, a parser description remains associated with its 'prototype', so that revisions of the prototype can be signaled to derivatives. Grammar convergence also covers this scenario, except that it cannot detect modifications that are gone after grammar extraction.

7 Concluding Remarks

If unit testing is the simple, pragmatic, and effective method to generally validate the I/O behavior of software modules, then grammar convergence is the simple, pragmatic, and effective method to keep scattered grammar knowledge 'in sync'. The method can also be used to capture and henceforth verify the differences between scattered grammar knowledge — both intended differences (due to evolution or implementational choices) and accidental differences (that cannot be resolved immediately). In one case study (see the authors' website), we have applied the method to the various grammars in the Java language specification; we have accurately captured the evolution from version to version, and we have spotted a substantial number of inconsistencies.

We currently work on the application of grammar convergence at the instance level (c.f., XML trees, derivation trees, parse trees, etc.) so that one can compare and converge 'data' from different software artifacts. Our implementation already supports some operators at the instance level so that instances of one grammar can be converted to instances of another grammar.

Our current implementation of grammar convergence does not infer transformation candidates in any way; the software engineer must use the output of grammar comparison intelligently. This is an obvious target for future work, and we expect useful input from other areas of software engineering: schema matching and data integration in the field of data modeling and databases [29]; comparison of UML models or metamodels in the context of model-driven engineering [34,7]; the computation of refactorings from different OO program versions [26,5].

Acknowledgement. The first author is grateful for opportunities to present and discuss some of this work at University of Waterloo, IEEE Kitchener-Waterloo, on August 6, 2008, at Dagstuhl Seminar 08331, 'Perspectives Workshop: Model Engineering of Complex Systems (MECS)', on August 12, 2008, and at invited talks at the METRIK Workshop, in Berlin, on November 21, 2008, and at the BENEVOL Workshop, in Eindhoven, on December 12, 2008.

References

1. Aho, A., Sethi, R., Ullman, J.: Compilers. Principles, Techniques and Tools. Addison-Wesley, Reading (1986)
2. Bouwers, E., Bravenboer, M., Visser, E.: Grammar Engineering Support for Precedence Rule Recovery and Compatibility Checking. ENTCS 203(2), 85–101 (2008)
3. Dean, T., Cordy, J., Malton, A., Schneider, K.: Grammar Programming in TXL. In: Proceedings of Source Code Analysis and Manipulation (SCAM 2002). IEEE, Los Alamitos (2002)
4. Dean, T., Cordy, J., Malton, A., Schneider, K.: Agile Parsing in TXL. Journal of Automated Software Engineering 10(4), 311–336 (2003)
5. Dig, D., Comertoglu, C., Marinov, D., Johnson, R.: Automated Detection of Refactorings in Evolving Components. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 404–428. Springer, Heidelberg (2006)
6. Duffy, E.B., Malloy, B.A.: An Automated Approach to Grammar Recovery for a Dialect of the C++ Language. In: Proceedings of 14th Working Conference on Reverse Engineering (WCRE 2007), pp. 11–20. IEEE, Los Alamitos (2007)
7. Falleri, J.-R., Huchard, M., Lafourcade, M., Nebut, C.: Metamodel Matching for Automatic Model Transformation Generation. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 326–340. Springer, Heidelberg (2008)
8. Hoare, C.A.R.: Proof of Correctness of Data Representations. Acta Informatica 1(4), 271–281 (1972)
9. Jin, D., Cordy, J., Dean, T.: Where's the Schema? A Taxonomy of Patterns for Software Exchange. In: Proceedings of International Workshop on Program Comprehension (IWPC 2002), pp. 65–74. IEEE, Los Alamitos (2002)
10. de Jonge, M., Monajemi, R.: Cost-effective maintenance tools for proprietary languages. In: Proceedings of International Conference on Software Maintenance (ICSM 2001), pp. 240–249. IEEE, Los Alamitos (2001)
11. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In: Proceedings of Generative programming and component engineering (GPCE 2006), pp. 249–254. ACM Press, New York (2006)
12. Kadhim, B., Waite, W.: Maptool—supporting modular syntax development. In: Gyimóthy, T. (ed.) CC 1996. LNCS, vol. 1060, pp. 268–280. Springer, Heidelberg (1996)
13. Kort, J., Lämmel, R., Verhoef, C.: The Grammar Deployment Kit. ENTCS 65(3), 7 Pages (2002); Proceedings of Language Descriptions, Tools, and Applications (LDTA 2002)
14. Kraft, N.A., Malloy, B.A., Power, J.F.: An infrastructure to support interoperability in reverse engineering. Information & Software Technology 49(3), 292–307 (2007)
15. Lämmel, R.: Grammar Adaptation. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 550–570. Springer, Heidelberg (2001)

16. Lämmel, R.: Grammar Testing. In: Hussmann, H. (ed.) FASE 2001. LNCS, vol. 2029, pp. 201–216. Springer, Heidelberg (2001)
17. Lämmel, R.: The Amsterdam toolkit for language archaeology. ENTCS 137(3), 43–55 (2004); Post-proceedings of the 2nd International Workshop on Meta-Models, Schemas and Grammars for Reverse Engineering (ATEM 2004)
18. Lämmel, R., Meijer, E.: Revealing the X/O Impedance Mismatch. In: Backhouse, R., Gibbons, J., Hinze, R., Jeuring, J. (eds.) SSDGP 2006. LNCS, vol. 4719, pp. 285–368. Springer, Heidelberg (2007)
19. Lämmel, R., Schulte, W.: Controllable Combinatorial Coverage in Grammar-Based Testing. In: Uyar, M.Ü., Duale, A.Y., Fecko, M.A. (eds.) TestCom 2006. LNCS, vol. 3964, pp. 19–38. Springer, Heidelberg (2006)
20. Lämmel, R., Verhoef, C.: Semi-automatic Grammar Recovery. Software—Practice & Experience 31(15), 1395–1438 (2001)
21. Lämmel, R., Wachsmuth, G.: Transformation of SDF syntax definitions in the ASF+SDF Meta-Environment. ENTCS 44(2) (2001); Proceedings of Language Descriptions, Tools and Applications (LDTA 2001)
22. Lohmann, W., Riedewald, G., Stoy, M.: Semantics-preserving migration of semantic rules after left recursion removal in attribute grammars. ENTCS 110, 133–148 (2004); Proceedings of 4th Workshop on Language Descriptions, Tools and Applications (LDTA 2004)
23. Malloy, B., Power, J., Waldron, J.: Applying software engineering techniques to parser design: the development of a C# parser. In: Proceedings of Conference of the South African institute of computer scientists and information technologists, pp. 75–82. ACM Press, New York (2002)
24. Moore, R.C.: Removing left recursion from context-free grammars. In: Proceedings of the first conference on North American chapter of the Association for Computational Linguistics, pp. 249–255. Morgan Kaufmann Publishers Inc., San Francisco (2000)
25. Morgan, C.: Programming from Specifications. Prentice Hall International, Englewood Cliffs (1990)
26. O'Keeffe, M., Cinnéide, M.O.: Search-based refactoring: an empirical study. Journal of Software Maintenance and Evolution 20(5), 345–364 (2008)
27. Parr, T.: The Reuse of Grammars with Embedded Semantic Actions. In: Proceedings of the 16th IEEE Conference on Program Comprehension (ICPC 2008), pp. 5–10. IEEE, Los Alamitos (2008)
28. Pepper, P.: LR Parsing = Grammar Transformation + LL Parsing. Technical Report CS-99-05, TU Berlin (1999)
29. Rahm, E., Bernstein, P.A.: A survey of approaches to automatic schema matching. VLDB Journal 10(4), 334–350 (2001)
30. Sellink, M., Verhoef, C.: Development, Assessment, and Reengineering of Language Descriptions. In: Proceedings of Conference on Software Maintenance and Reengineering (CSMR 2000), pp. 151–160. IEEE, Los Alamitos (2000)
31. Sirer, E., Bershad, B.: Using Production Grammars in Software Testing. In: USENIX (ed.) Proceedings of Domain-Specific Languages (DSL 1999), pp. 1–13. USENIX (1999)
32. Stevens, P.: Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 1–15. Springer, Heidelberg (2007)
33. Wile, D.: Abstract syntax from concrete syntax. In: Proceedings of International Conference on Software Engineering (ICSE 1997), pp. 472–480. ACM Press, New York (1997)
34. Xing, Z., Stroulia, E.: Refactoring Detection based on UMLDiff Change-Facts Queries. In: Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006), pp. 263–274. IEEE, Los Alamitos (2006)

Application of Graph Transformation in Verification of Dynamic Systems[⋆]

Zarrin Langari and Richard Trefler

David R. Cheriton School of Computer Science
University of Waterloo, Canada
{zlangari,trefler}@cs.uwaterloo.ca

Abstract. A communication system evolves dynamically with the addition and deletion of services. In our previous work [12], a graph transformation system (GTS) was used to model the dynamic behaviour of a telecommunication system. In this paper, we show how GTS modeling can facilitate verification of invariant properties of potentially infinite-state communication systems. We take as a case study for this approach an invariant property of telecommunication service components that can act both as the source and the target of a connection. Verifying an ordering among service components to be invariant is essential to guarantee the desirable behaviour of these services. We show how the verification can be performed by the analysis of a finite set of transformation rules describing the GTS system model. We prove that invariant properties are preserved in a GTS model if the set of transformation rules describing the model satisfies the property. Thus, we show how to perform system verification through analysis of the model description without building the full system state space.

1 Introduction

Connection-oriented communication protocols require a connection session to be established prior to data transfer. In these protocols, services are described by the inter-operation of individual units of functionality known as *features*. Dynamic evolution of a communication system changes the topology of the system as new features are added to or existing ones deleted from the connection session. This dynamic evolution may cause the violation of inter-feature specifications, thereby causing undesirable feature interactions.

In the current work, we show how to verify high-level invariant properties of connection-oriented services given as graph transformation system (GTS) [6,14] model descriptions. We show that if a property is preserved by the finite set of transformation rules describing the system model, and if the initial state satisfies the property, then the property is an invariant of the protocol. Therefore, our verification method avoids the explicit analysis of the behaviours and potentially enormous state space that the transformation rules encode.

[⋆] The authors' research is supported in part by a grant from the Natural Sciences and Engineering Research Council of Canada.

Motivation. Among connection-oriented services, some so called *reversible* modules perform crucial functions such as controlling multi-point connections and transferring the connection from one endpoint device to another endpoint device without disturbing an ongoing connection [18]. Telecommunication protocols are perhaps the best examples of connection-oriented systems, and in fact, reversible modules in these systems are able to initiate a call on their own. For instance, after a connection has been dropped due to a failure, such a reversible module (feature) will re-establish the connection to the appropriate endpoint. As instances of reversible modules in telephony, we can name Call-Waiting for switching between two incoming calls; Automatic-Call-Back that offers the caller an automatic call whenever the callee is idle, in case of call failure (e.g. busy line); and Mid-Call-Move that moves an endpoint of a connection from one device to another while the caller and the callee are in the "talk" mode. Reversible modules have an important role in maintaining source and destination symmetry. This symmetry ensures that messages are not being lost and that the connection is long-lasting and continuous. This continuity is characteristic of well-behaved communication protocols.

A strong motivation for the current work is to find a way of managing interactions among reversible service modules by imposing an appropriate ordering on them that satisfies the system specification and governs desirable interactions.

Contribution. Distributed communication protocols are notoriously resistant to formal modeling due to their size and complexity. In our previous work [12], we used graph transformation, a straight forward, visual formalism to do this modeling. In experiments with the Distributed Feature Composition (DFC) protocol[10], an IP-based telecommunication architecture of AT&T, we found that graph transformation offers several key advantages over naive methods in modeling the dynamic evolution of a reactive communication protocol. The structure of the generated model closely resembles the way in which communication protocols are typically separated into three levels: the first describing local features or components, the second characterizing interactions among components, and the third showing the evolution of the component set. The graph transformation semantics follows this scheme, enabling a clean separation of concerns when describing a protocol.

In addition, in distributed communication protocols, verifying properties of feature compositions is problematic due to the state explosion problem and may not even be decidable. Therefore, avoiding explicit analysis of the system state space is desirable. In this paper, we address the verification problem for a class of systems with potentially unbounded state spaces. We show that an invariant system property encoded as a graph can be verified against the GTS model [12] by examining the model's finite set of transformation rules, and without resorting to the exploration of the full state space. While the problem is in general undecidable, we show that the property is satisfied by the GTS model if the set of rules are property preserving. To enable this type of verification, first we define the notion of graph satisfaction. Then we use this notion to define what it means for a transformation rule to preserve a property.

We then show that the ordering property of reversible features in communication systems is guaranteed through our analysis of the GTS model of the system. We choose DFC as our communication system case study. We also present the invariant ordering property of reversible features in DFC. This property states that the sequence of reversible features in an existing call associated with an address is an invariant of the call. Maintaining this property is important to avoid feature interaction problems due to the dynamic evolution of the system.

Structure of the Paper. In Section 2, an informal description of DFC semantics and the specifications of the ordering property of reversible features are presented. An overview of GTS modeling is given in Section 3 and is followed by a description of the verification problem and its analysis in Section 4. In Section 5, we present the verification of the ordering property. Related work is presented in Section 6, and we conclude in Section 7.

2 Motivating Problem: Reversible Features in DFC

2.1 Introduction to DFC

In this section we define DFC [10] terminologies used for building a connection.

Usages. In DFC, a request for telecommunication service is satisfied by a *usage*, which describes the dynamics of a telephone call between two or more end parties. In our modeling of DFC [12], a usage is presented visually as a graph of *boxes* and *internal calls*. A *box* is a concurrent process providing either feature functions (*a feature box*) or presenting an end party (*an interface box*). An internal call is a basic connection between any two points (phone-to-phone, phone-to-feature, feature-to-phone and feature-to-feature) without any intervening feature. Each feature box is context-independent, so feature boxes can be added, deleted, and changed. Each interface box has an *address* which is a string used to identify a telecommunication device attached to a network. Each feature box has a *box type* that corresponds to the feature that it implements, and an address. A feature box is instantiated from its box type on behalf of an address.

For each address, the sequence of box types that should be assembled into a usage on behalf of the address when the address is the source or the target of a connection are called *Source* or *Target Subscriptions*. There is a partial precedence order between both source-subscribed and target-subscribed box types of an address. This order is an input to the routing algorithm and is chosen by the designer based on features priorities to eliminate undesirable feature interactions. Figure 1.a shows a straightforward usage formed when the device with address X that has subscribed to features $F1$ and $F2$ requests a connection to end party Y that has subscribed to features $F3$ and $F4$.

Routing. When an interface box requests a connection, the connection is built using a DFC routing algorithm. This algorithm [10] has three basic methods: *new*, *continue*, and *reverse* that are executed on behalf of interface or feature

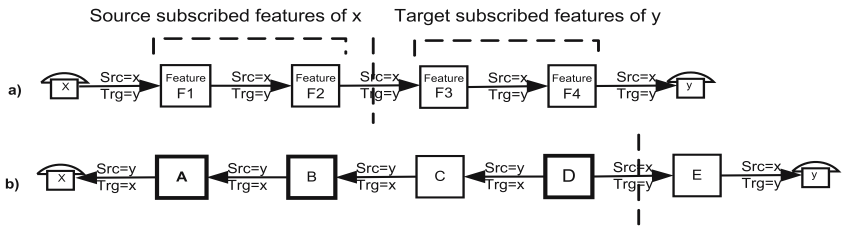


Fig. 1. a) A simple usage. b) A usage with reversible boxes A, B, D.

boxes. The method *new* is executed by an interface box that requests the connection and creates a setup signal. The method *continue* is applied to a received setup signal by a feature box and results in the resubmission of that signal to the next box closer to the desired end party. The method *reverse* is performed by a feature box to reverse its incoming call, i.e. change the direction of a call, possibly back towards the initiating interface box. For routing, the router chooses feature boxes in sequence from a list, called *route*, associated with the current address. The route consists of a sequence of source-subscribed box types of the source address and a sequence of target-subscribed box types of the target address. The point on the connection path of the usage in which the call is routed from source to target-subscribed boxes, is called the *mid-point* of the call.

If feature boxes use only the continue method, then the simplest usage as depicted in Figure 1.a has instances of all source-subscribed box types of the source address, followed by all target-subscribed box types of the target address. A usage with activated reversible features may be formed in an unusual way with internal calls in different directions. As an example, consider the usage in Figure 1.b, with features A, B, C, and D subscribed to by address X from which A, B, and D are reversible (depicted in bold), and feature D is activated.

2.2 Invariant Ordering Property of Reversible Features

The invariant ordering property of reversible features is significant in a usage because it guarantees that the interactions of all the reversible features follow a desired pattern even though the usage has changed from its initial configuration [16]. In this section, we explain this property using DFC specifications.

Using the informal description of DFC specifications given in the previous section, we have extracted three specification statements as first-order logic predicates to describe the above mentioned property about orderliness of reversible features in DFC routing. In these statements, a is an address from the set of existing addresses, $Addr$; $Reversible$ is the set of reversible feature boxes subscribed to by an address; $SrcSubscriptions$ and $TargetSubscriptions$, respectively, show the sequences of source and target subscriptions of an address. $Precedes(\prec)$ is a relation between feature boxes that shows the order in which they are assembled in the usage of an address. R and R', respectively, are the projection of source and target subscriptions onto reversible boxes. Therefore, for address a we have:

$$a.R = a.Reversible \cap a.SrcSubscriptions \quad a.R' = a.Reversible \cap a.TrgSubscriptions$$

In the box below, we have the specification statements about reversible features. In these specifications, angle brackets " $<$ " and " $>$ " are used to denote the list of members in a sequence. In order to show that a sequence is a subsequence of another sequence we use the notation $\sqsubseteq$, e.g. $A \sqsubseteq B$ denotes that sequence A is a subsequence of sequence B. The first specification in the box states that if an address subscribes to a reversible box, it must subscribe to it both as a source and as a target address. The second one states that the precedence order relation ($\prec$) in the set of reversible boxes being subscribed to as source subscriptions of an address is a total order. In this specification, if R is replaced by R' then we get the total order for reversible boxes being subscribed to as target subscriptions. The third specification expresses that the subsequences of source and target reversible features of an address have an opposite precedence order.

1. $(\forall a \in Addr, \forall f \in a.Reversible) \Rightarrow$
   $(f \in a.SrcSubscriptions \Leftrightarrow f \in a.TrgSubscriptions)$

2. $(\forall a \in Addr, \forall f_1, f_2 \in a.R, f_1 \neq f_2) \Rightarrow$
   $(((f_1 \prec f_2) \vee (f_2 \prec f_1)) \wedge \neg((f_1 \prec f_2) \wedge (f_2 \prec f_1)))$

3. $(\forall a \in Addr, \forall f_1, f_2 \in a.Reversible, f_1 \neq f_2) \Rightarrow$
   $((< f_1, f_2 > \sqsubseteq a.SrcSubscriptions) \Leftrightarrow (< f_2, f_1 > \sqsubseteq a.TrgSubscriptions))$

The third specification is needed to ensure that a usage consisting of multiple features is constructed in an acceptable way, and in particular to ensure that a new usage does not need to be established when the direction of signals is reversed. Using the above specifications, we have this property:

Requirement. *In a usage associated with an address, if we order the feature boxes from outermost (closest to the endpoint) to innermost (closest to the midpoint), the sequence of reversible feature boxes associated with the address is an invariant sequence regardless of how the usage is initially constructed.*

To justify the invariant ordering we use an example that shows an undesirable behaviour when reversible features do not follow an invariant ordering. The example in Figure 2 shows that a usage has been set up between a source x with subscribed reversible features Call Waiting (CW), 3-Way Calling($3WC$), and Automatic Call Back (ACB), and the end party y. We are not concerned with the subscribed features of end party y for this example. Scenario 1 in Figure 2 demonstrates a usage of address x with the source ordering: $CW \prec 3WC \prec ACB$. Automatic Call Back acts as a source feature, and if an outgoing call fails, it offers the user x a chance to activate the feature. If x does so, the ACB

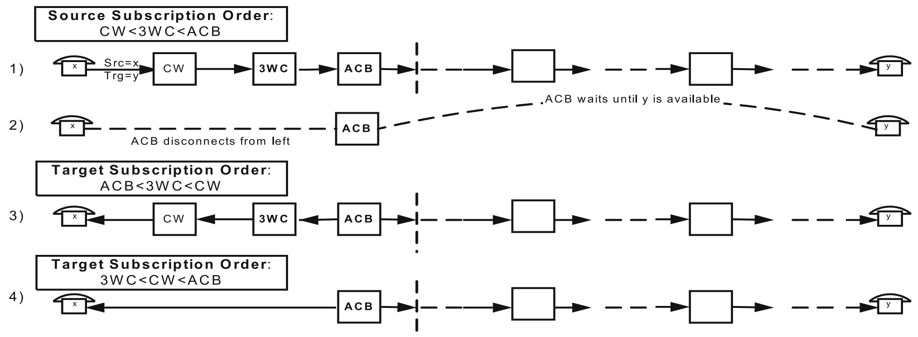

Source Subscription Order:
CW<3WC<ACB
1)
Src=x
Trg=y
CW
3WC
ACB
2)
ACB
ACB disconnects from left
ACB waits until y is available
Target Subscription Order:
ACB<3WC<CW
3)
Target Subscription Order:
3WC<CW<ACB
4)
x
y

Fig. 2. Interaction of three reversible features in a usage

box disconnects from $3WC$, and waits until the original callee is available, this behaviour is depicted in Scenario 2 of Figure 2, though we are not concerned with the details of how ACB knows the callee is available. Once y becomes available, as depicted in Scenario 3 of Figure 2, ACB places a call to x using reverse and the target ordering $ACB \prec 3WC \prec CW$. When it is connected to x, it places a (hopefully successful) call to the original callee. The call to the callee is a continue rather than a reverse.

Now, say, there is an error in the target order and we have the order as $3WC \prec CW \prec ACB$. As shown in Scenario 4 of Figure 2, because of the incorrect order, ACB is the last feature in the order, and $3WC$ and CW are not routed to. Therefore, if x is actually engaged in another call via CW, it will miss the call from ACB, even though CW would have enabled it to take the call.

We conclude that the reversible sequence associated with an address must have a fixed ordering from the left to the right. In the following sections, we model this property as a graph and show how to verify it via GTS modeling.

3 Graph Transformation System Modeling

GTS is a powerful formalism for modeling the semantics of distributed systems [6]. In our previous work [12], we modelled the dynamic behaviour of DFC as it changes over time. In this modelling, we use a graph transition or a graph transformation system, in which nodes represent states of the system at a particular point and transitions show how the system evolves from one state to the next. Each state of the system is modelled as a usage graph of features and interface boxes and by using the graph transformation system we describe how a usage changes its state to another by the application of transformation rules. The GTS model that we use to describe DFC is defined in this section.

Definition 1 (Graph [8]). *A graph $G = (V, E, Src, Trg, Lab)$ consists of a set V of nodes, a set E of edges, and functions $Src, Trg : E \rightarrow V$, and the labelling function $Lab : E, V \rightarrow lab$, where lab belongs to a set of labels.*

Definition 2 (Graph Morphism [14]). *A graph morphism $f : G \to H$ maps nodes (V) and edges (E) of graph G to nodes and edges of graph H. $f_v : V_G \to V_H$ and $f_e : E_G \to E_H$ are structure-preserving functions, that is, we have for all edges $e \in E_G$, $f_v(Src_G(e)) = Src_H(f_e(e))$, $f_v(Trg_G(e)) = Trg_H(f_e(e))$, and $Lab_H(f_e(e)) = Lab_G(e)$. If f_v, f_e are total functions, then we have a total morphism, and if these functions are partial, we have a partial morphism.*

Note that in a structure-preserving mapping, the shape and the edge labelling of the original graph are preserved. A graph morphism $f = (f_e, f_v)$ is called injective if both f_e and f_v are injective mappings.

Definition 3 (Graph Transition System [14,9]). *A transition system is defined as: $\mathcal{G} = \langle S, T, I \rangle$. S is a set of states (nodes), where each state has a graph structure, I is an initial state, and T is a set of transitions based on graph morphism: $T \subseteq S \times P \times S$ where P is a set of transformation rules. A transformation t is defined as $t : G \overset{r}{\Longrightarrow} H$ where $r \in P$. The transformation rule r appears in the form: $L \to R$, L is the left side graph and R is the right side graph of the rule. The application of a rule r to a graph G, is based on a total morphism between L and graph G. We write $t : G_0 \overset{r}{\Longrightarrow} G_1$ to show that the system will be transformed from the state G_0 to G_1 by the application of rule r. A graph transformation $t : G_0 \overset{*}{\Longrightarrow} G_n$ is a series of $G_0 \Rightarrow G_1 \Rightarrow ... \Rightarrow G_n$ direct graph transformations.*

In some cases, the application of rules is restricted by conditions. These conditions forbid specific graph elements to be in a graph for a rule to be applicable.

These conditions are just present in the left side graph (L) of a rule, since the application of a rule to a graph G is based on morphisms between G and L. In general, the process of applying a rule to a graph is that everything in the left side graph (L) but not in the right side graph (R) will be *deleted*, everything in R which is not in L will be *created*, and everything that is in both sides will be *preserved* [14]. A total match between the left side subgraph of a rule and a subgraph in the source graph is made, then the source subgraph is deleted and replaced by the right side subgraph R.

For graphs, we adopt a GTS model with attributed graphs and node identification [14,4,12]. In some cases nodes or edges of a graph are identified by data attributes such as strings and numbers. The GTS defined above supports attributed graphs. In this model, every node of the graph is unique based on its attributes. In the definition of GTS we also require the application of transformation rules based on injective mappings which means we have injective mappings between the left side of the rule and the source graph.

4 Verification through GTS

We define the verification problem of a GTS and present our method to solve it. In [11], Kastenberg and Rensink showed that a labelled transition system (LTS) created by a graph transition system, can be seen as a representation of a Kripke

Structure for a system model. Therefore, to verify an invariant property on a generated LTS, all the reachable states should be verified for the satisfaction of the property. GTSs can generate an infinite-state model; hence, automatic verification is problematic for these systems. At an appropriate level of abstraction, communication protocols are typically modelled as infinite-state systems. Thus, it is reasonable to model them using GTSs.

Here we address the verification of invariant properties that are expressed by the CTL modality AG and atomic propositions [9] modelled as graphs called *proposition graphs*. Unlike the work in [5] that is restricted to forbidden (negative) constraints, we consider positive propositions, because in most cases negative constraints can be expressed by negation of positive constraints [8].

We use the notation of a graph transformation tool, GROOVE [11], for expressing proposition graphs with labels as regular expressions (e.g. Kleene star labels). We use these labels to compactly express feature connectivity patterns, for instance, to show that between two features of interest there may be an arbitrary length sequence of intervening features. Therefore, we need to extend the definitions of graph and graph morphism to include Kleene star labels. The following definition of *regular expression graph* provides this extension. A regular expression graph is a graph in which edges may be labelled with the Kleene star operator over the set of labels. Using regular expression graphs in the proposition graphs and transformation rule graphs makes these graphs more expressive.

Definition 4 (Regular Expression Graph (REG)). *An REG is a graph G where $\exists e \in E_G$ such that $Lab_G(e) \in Labels^*$.*

An REG can be a subgraph of the left side or the right side graph in a transformation rule or a subgraph of a proposition graph. Examples of these two cases are depicted in Figures 4 and 5 respectively. There is an exception that Kleene star labels are not allowed on newly created graph edges (on the right side of a rule) or on edges to be deleted (on the left side of a rule.)

Definition 5 (Path). *In a graph $G = (V, E, Src, Trg, Lab)$, a path p from node v_1 to node v_n is a sequence of nodes connected by edges: $p = \{v_1, v_2, ..., v_n\} \in V$ such that $\{e_1, e_2, ..., e_{n-1}\} \in E$, $v_1 = Src(e_1)$, and for all $1 \leq i \leq n-2$, $v_{i+1} = Trg(e_i) = Src(e_{i+1})$, and $v_n = Trg(e_{n-1})$, and $Lab(e_i) \in Labels^*$.*

In a path, if there are Kleene star labelled edges, then G is an REG and the path is defined for an REG. Hence, the sequence s of edge labels in path p, written as $s = (Lab(e_1)...Lab(e_{n-1}))$, specifies a language. For example, for a path p, with consecutively labelled edges a, x^*, y^* and b, the sequence $s = (ax^*y^*b)$ specifies the language of path p, and the string $w = axb$ is a member of that language. To specify the satisfaction of a proposition graph (an REG) by another graph (possibly an REG), we need to define a specific type of morphism to map a path in one REG to another path in the second REG.

Definition 6 (Regular Expression Graph Morphism). *$rgm : G \to H$ is an REG morphism between graphs G and H, if either G or H is an REG and for a path $p = \{v_1, ..., v_n\}$ in G there is a path $q = \{u_1, ..., u_n\}$ in H such that:*

- *There is a graph morphism $m : V_G \rightarrow V_H$ between the beginning and the end nodes of these two paths, written as $u_1 = m(v_1)$, $u_n = m(v_n)$.*
- *For $2 \leq i \leq n-1$, three cases may occur:*
  1. *If both G and H are REGs, then the language specified by the sequence of corresponding labels over the edges connecting nodes v_i in p is a subset of the language specified by the sequence of labels over the edges connecting nodes u_i in q.*
  2. *If H is an REG, and G is a graph without Kleene star labelled elements, then the string specified by the sequence of corresponding labels over the edges connecting nodes v_i in p is a member of the language specified by the sequence of labels over the edges connecting nodes u_i in q.*
  3. *If G is an REG, and H is a graph without Kleene star labelled elements, then this is similar to case 2 with respective changes for G and H.*

We have total or partial REG morphisms, if the mappings are respectively total (for the set of non-overlapping paths in G) or partial.

The verification problem is solved using a forward analysis method which ensures that the invariant property is never violated by the application of transformation rules. Thus the required invariant is satisfied by the system model. Though the example we picked here for the invariant property is an ordering property of DFC, the method can be used for invariants that are expressed by positive proposition graphs. For example, in a linear linked list we may wish to verify that always all nodes of the list contain some data value. To express the verification problem of invariant properties we define our notion of graph satisfaction:

Definition 7 (Graph Satisfaction). *An REG or a graph G satisfies an REG or a graph ϕ, $G \models \phi$, iff there exists an appropriate total morphism (graph or REG morphism) m between ϕ and G written as $m : \phi \rightarrow G$. A graph G weakly satisfies a graph ϕ, written as $G \models^w \phi$, iff there exists a non-empty appropriate partial morphism (graph or REG morphism) between ϕ and G, $m^w : \phi \rightarrow^w G$.*

Following this definition, then $G \not\models^w \phi$ (G does not weakly satisfy ϕ) means there is no morphism between ϕ and G, and it implies that G does not satisfy ϕ. Note that empty morphisms where there is no mapping between two graphs are in fact types of partial morphism, but they are excluded from the definition of $G \models^w \phi$; therefore, empty morphisms are considered to be non-satisfying. Figure 3 shows a graph satisfaction example based on morphisms in Definition 6.

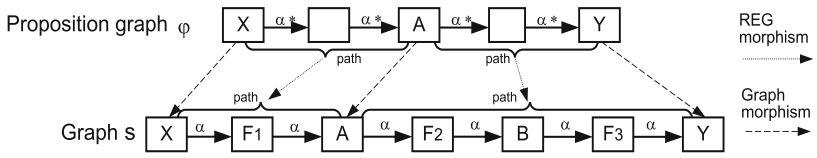


Fig. 3. A proposition graph φ is satisfied by graph s

Proposition graphs may be defined as boolean combinations of two or more graphs. Then graph satisfaction is defined as:

$$G \models \varphi \vee \psi \textit{ iff } G \models \varphi \textit{ or } G \models \psi$$
$$G \models \varphi \wedge \psi \textit{ iff } G \models \varphi \textit{ and } G \models \psi$$

Using above definitions, the following theorem states a satisfying condition for a transformation rule to preserve a property ϕ. In general, when the property ϕ is satisfied in a state, the transformation rule preserves the property, if this rule does not transform the state graph in a way that violates the property. That is, given a property ϕ as a proposition graph, a rule r is guaranteed to *preserve* ϕ if its left side graph does not weakly satisfy ϕ or its right side satisfies ϕ.

Theorem 1 (Property Preservation). *Given $\mathcal{G} = \langle S, T, I \rangle$, let ϕ be a state property, and $\langle s, r, t \rangle \in T$, where r is a rule, $r : L \rightarrow R$, and $s, t \in S$, and suppose $s \models \phi$. If $L \not\models^w \phi$ or $R \models \phi$, then $t \models \phi$.*

Proof. Based on Definition 3, when a rule is applied to a state graph s, it reconfigures s resulting in another state graph t. Considering the fact that both s and ϕ are graphs, if s already satisfies the property ϕ, and if r does not reconfigure parts of s that have a mapping to ϕ, then the resulting graph satisfies ϕ.

To make sure that a rule preserves a property in the transformed state, we have to check how the transformation affects the existing mapping of ϕ to s. Thus the morphisms between the left and the right side graphs of r and ϕ determine if r affects ϕ's mapping to s or not. In other words, based on Definition 7, how L and R satisfy ϕ leads us to find out if r's transformation process violates ϕ. Though there might be different types of satisfaction relation between L, R and ϕ resulting in different combinations, all those combinations can be summarized in two cases, which we prove separately:

1) $s \models \phi$ and $R \models \phi \Rightarrow t \models \phi$
2) $s \models \phi$ and $L \not\models^w \phi \Rightarrow t \models \phi$

Case 1: This case states that if the right side of a rule satisfies a property, no matter what is the satisfaction situation with its left side, the rule's application preserves the property. Based on the rule's application process defined in Section 3, the common elements in R and L are preserved and other elements in R are created in the transformed state. Therefore, if R satisfies ϕ, then ϕ's elements will be mapped to the transformed state, no matter if L satisfies ϕ or not.

Case 2: In this case, when $L \not\models^w \phi$, it means there is no morphism (partial or total) between L and ϕ. Since $s \models \phi$ and there is a total graph or REG morphism between L and s (Definitions 3, 4), then whether R satisfies ϕ or no, either r transforms s with creation of some or all elements of ϕ, or it transforms a part of s that does not map to any elements of ϕ. □

The second case occurs when a rule transforms parts of a big graph that do not interfere with the proposition graph elements, in other words, those parts do not have a mapping to the proposition graph. In all other cases ($L \models \phi$ and $R \not\models^w \phi$, $L \models \phi$ and $R \models^w \phi$, $L \models^w \phi$ and $R \models^w \phi$), the rule may transform the graph in a way that elements mapping to the proposition graph in the left side are deleted in the transformed state. These cases may therefore violate the property. Thus, such transformation rules may or may not preserve the property. Therefore, this verification technique is sound but incomplete (may introduce false negatives).

Note that this theorem may not hold for GTSs without node identification[14], where we have similar nodes with the same attributes, or when rules are applied based on non-injective mappings. For instance, consider the non-injective mapping of the left side of a rule to a graph s where there are two sets of mappings from L to a subgraph in s. If both sides of the rule and also s preserve a property (based on the same subgraph mapping), and rule applies to s with the goal to delete the mapped subgraph, because of the priority of deletion over preservation in the rule application approach [12], the transformed state graph does not satisfy the property. This exception does not occur in our GTS rules built to present DFC behaviour, because we use graph nodes with identification and injective graph morphisms for the rules application in our modelling.

The above Theorem is used to prove the satisfaction of invariant properties defined as a CTL formula $AG(\phi)$ in a GTS description of a system.

Corollary 1. *For the graph transition system $\mathcal{G} = \langle S, T, I \rangle$ and the state property ϕ, $\mathcal{G} \models \phi$, if $I \models \phi$ and for all rules $r \in \mathcal{G}$, r preserves ϕ.*

Proof. This Corollary can be proved by induction using $I \models \phi$ as the base case. Based on Theorem 1, a hypothesis is made to assume that the property is preserved by all the rules in $\mathcal{G}$ that are applied to the states up to the kth level in the LTS. In the induction step, since we have proven that the property is satisfied at the states of the LTS in level k, and because all rules are property preserving, then the given property also holds through all the rules application at level $k + 1$ of the LTS. □

5 Verifying an $AG(\phi)$ Property in a DFC Usage

After the establishment of a connection in a DFC usage, the usage can grow, shrink, split, or merge with other usages. The establishment of a connection is based on DFC routing algorithm, which builds a connection from the ordered feature list associated with an address. Each feature box requests a connection using the “new”, “continue”, or “reverse” methods (Section 2.1).

The DFC routing algorithm is described by two GTS models in the following steps. In the first model, a GTS is used to build a connection for a usage associated with two endpoints. These GTS transformation rules describe the methods “new” and “continue” found in the DFC routing algorithm. For interested readers, the GTS model for establishing a connection was given in [12]. The second GTS includes those transformation rules for activation of caller’s and callee’s reversible features. From these rules, caller’s reverse activation rule is illustrated in Figure 4. The other rules for propagation of reverse signal to the source of the connection and callee’s reverse activation rules are similar, thus omitted here.

After the establishment of a connection, which we know satisfies the ordering property that is given explicitly in the model description, the second graph transition system, which utilizes the built connection as the initial state, encodes dynamic changes of the usage graph. As an example, Figure 4 illustrates the rule for activation of a caller's subscribed reversible feature. In this picture, nodes labelled with $?x$ and $?y$ match an arbitrary label for the caller and callee names respectively. Edges labelled with $?x, ?y$ show that the edge has the same labelling as caller, callee names pair and present that each segment of the path is related to which caller and callee. $?x*, ?y*$ edge labels express that two features are connected through a sequence of intervening features connected by x, y-labelled edges. Features have several attributes such as their name; subscriber; status, which shows if it is a source subscription or target subscription or both; and mode, which shows whether a feature is reversible. Attributes are depicted as circular nodes, and elements in the left side of the rule that are being deleted are illustrated as thin dashed elements. The solid fat elements show those elements that are being created by the right side of the rule, and the rest of elements are those preserved by both sides of the rule. To express that the reversible feature F in Figure 4 is connected to a callee through a sequence of non-reversible features, Kleene star labelled edges are used (REG). This figure shows the activation of a caller's reversible feature.

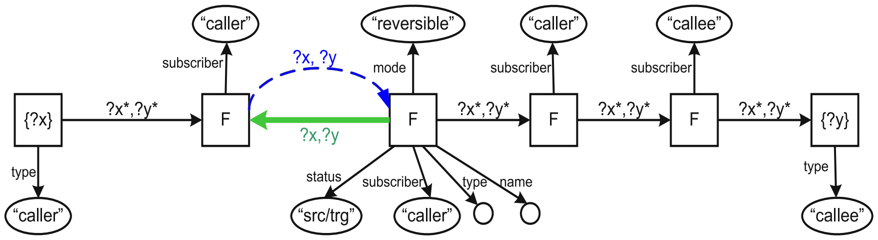


Fig. 4. The reverse rule ActivateCallerReverseFeature

We verify the invariant $AG(\phi)$ in DFC usages, where ϕ denotes the correct ordering of reversible features of an address. We use the requirement in Section 2.2 and REG definition to encode property ϕ as proposition graphs connected by operators $\vee$ and $\wedge$. The first case to consider with any usage is when reversible features have been assembled into the usage, but are not active. Therefore, a proposition graph is needed to ensure the ordering of these features as they appear in the established connection path of an address. The next step is to construct the proposition graphs that ensure the correct ordering of reversible features after they are active. For this step, we use the fact that there is a total precedence order relation among these features, and this ordering for source subscribed features is the opposite of the ordering for the target subscribed features of an address. Thus, when any of these features activates, it should be verified that only the direction of signals is changed, because the role of the reversible feature has been changed from source to target or vice versa. So these proposition

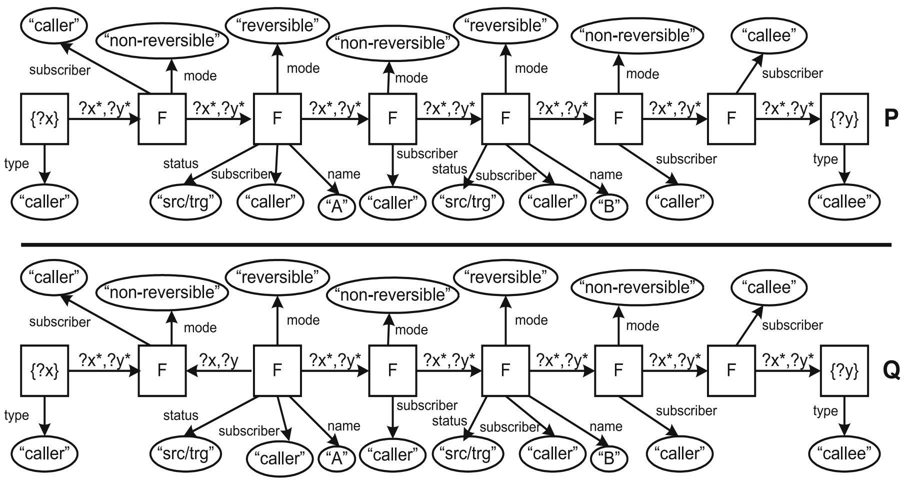


Fig. 5. Propositions P, Q in the property $AG(\phi)$

graphs consider the correct activation of each reversible feature, ensuring that the existing established connection path is used.

The invariant ordering property of two reversible features, A and B, subscribed to by a caller in an established connection between the caller and a callee in a DFC usage is stated as $AG(\phi)$. Two of proposition graphs that participate in constructing ϕ are P and Q as depicted in Figure 5. In this picture, proposition P shows that there is an established connection between the caller and the callee ensuring that the reversible features A and B appear in the connection in an order, with zero or more non-reversible features in between (shown as Kleene star labelled edges). In this example, we are not concerned with the callee's subscribed features, so these features are abstracted as an F labelled node connected by zero or more $?x, ?y$ labelled edges. Proposition Q shows that if the reversible feature A activates, the order of B respective to A does not change in the established connection path from the caller to the callee. Though other propositions are omitted here, they are similar to Q for checking the propagation of reverse signal to the caller and the activation of reversible feature B. In GROOVE [11], propositions are specified in the form of transformation rules with identical left and right sides.

Now we state the verification problem for invariant ordering of reversible features in a DFC usage in Lemma 1.

Lemma 1. *Let $\mathcal{G} = \langle S, T, I \rangle$ be the DFC transition system, and I be the state of an established connection associated with address A. Let ϕ be the invariant ordering property of reversible features associated with address A, such that $I \models \phi$. If for all rules r in $\mathcal{G}$, r preserves ϕ, then $\mathcal{G} \models \phi$.*

The Lemma is easily proved using Corollary 1 to show that the ordering property in an established connection of a DFC usage can be verified through our GTS model of DFC. The GTS with an established connection as the initial state is given as a transition system $\mathcal{G} = \langle S, T, I \rangle$, where I is an established connection state. We have developed $\mathcal{G}$ with a small, but important representative set of transformation rules that describe DFC behaviour for reversible features, and all are property preserving based on Theorem 1. After the connection has been set up between two endpoints, regardless of how the usage graph evolves, the ordering sequence of reversible features associated with any of the addresses at each end of the connection should remain a fixed sequence.

The number of rules that describe the behaviour of DFC are finite. Thus, we are able to verify invariant properties on a potentially infinite state system.

6 Related Work

There are two main types of verification work using GTSs. The first type focuses on the verification of finite-state systems [15,7,11]. In the current work, which is a sound though incomplete technique (discussed in Section 4), we focus on dynamic systems that are not a priori finite state.

Similar to our work, the second type considers verification of infinite-state systems. Interesting work by Baldan, Corradini, and König [1,2,3] considers an abstraction approach through unfolding of a GTS by means of constructing finite Petri-net structures. This work provides an approximate technique and may introduce false positives [1,2,3]. In addition, in this approach, properties are not formulated graphically.

The work in [13] also considers an abstraction approach called shaping that partitions elements of a graph based on their similarities. This work uses an alternate GTS formalism that does not make use of graph morphisms utilized in the current work. Furthermore, the approach of [13] is not precise, because concretization of the same abstract graph is different in shape and structure.

The work in [9] focuses on assertional reasoning and constructing a weakest precondition, and though it is applicable to infinite-state systems, verifying invariants using this approach is difficult. In [9], properties are graph morphisms based on rule application conditions, while in our approach, properties are graphs that can be generated in a straightforward manner from user requirements.

The work in [5] also addresses the verification of changing structure modelled by GTSs. This work is an approximative invariant verification method, and it only considers the conjunction of unsafe situations where the proposition graphs encode forbidden cases. Unlike our work that is a type of forward analysis of rules against the requirement graph, the work in [5] does the verification using a backward reachability analysis. The work analyzes the actual system states that are unsafe by explicitly applying rules backward to these states. This work uses safe system states to build a requirement, whereas invariant requirements are often given by the user and can be built directly as a proposition graph.

7 Conclusion

In [12], it is shown that graph transformation is a powerful formalism for modelling the dynamic evolution of communication and telecommunication systems.

In the present work we show a verification method for invariant checking on the GTS model of a telecommunication system. GTSs generate an LTS as the transition graph. The generated LTS for a communication system may lead to an infinite state model, and verifying invariants on the model requires analysis of the full state space.

Using the graph satisfaction notion we address the verification of an invariant using the finite set of transformation rules that describe the system behaviour. Further, we present the conditions under which a transformation rule preserves a property. Then we show that if the initial state of the GTS model satisfies the property, and if all transformation rules are also property preserving, then the system satisfies the property. Therefore, by analyzing the transformation rules, we are able to verify a property without explicit exploration of the state space.

This behaviour is true for invariants or properties of the form $AG(\phi)$. The proposition ϕ is described as a graph and we define two forms for transformation rules and show that, if all transformation rules are of one of those two forms, then they are ϕ preserving. For future work, we are implementing a utility that checks the preservation of a property by analyzing a transformation rule. Also, we would like to consider more complex paths and path properties in address translation cases, where reversible features change their source or target address [17]. We are also considering verification of liveness and fairness properties.

Acknowledgement

We are grateful to Pamela Zave, Shoham Ben-David and Ali Taleghani for their many insights and constructive comments on this research.

References

1. Baldan, P., Corradini, A., König, B.: A static analysis technique for graph transformation systems. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 381–395. Springer, Heidelberg (2001)
2. Baldan, P., Corradini, A., König, B.: Verifying finite-state graph grammars: an unfolding-based approach. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 83–98. Springer, Heidelberg (2004)
3. Baldan, P., König, B., Rensink, A.: Summary 2: Graph grammar verification through abstraction. In: Dagstuhl Seminar Proceedings, vol. 04241 (2005)
4. Baresi, L., Heckel, R.: Tutorial introduction to graph transformation: A software engineering perspective. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 402–429. Springer, Heidelberg (2002)
5. Becker, B., Beyer, D., Giese, H., Klein, F., Schilling, D.: Symbolic invariant verification for systems with dynamic structural adaptation. In: ICSE 2006, pp. 72–81 (2006)
6. Degano, P., Montanari, U.: A model for distributed systems based on graph rewriting. J. ACM 34(2), 411–449 (1987)
7. dos Santos, O.M., Dotti, F.L., Ribeiro, L.: Verifying object-based graph grammars. Software and System Modeling (3), 289–311 (2006)

8. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer, New York (2006)
9. Habel, A., Pennemann, K., Rensink, A.: Weakest preconditions for high-level programs. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 445–460. Springer, Heidelberg (2006)
10. Jackson, M., Zave, P.: Distributed feature composition: A virtual architecture for telecommunications services. Software Engineering 24(10), 831–847 (1998)
11. Kastenberg, H., Rensink, A.: Model checking dynamic states in GROOVE. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 299–305. Springer, Heidelberg (2006)
12. Langari, Z., Trefler, R.: Formal modeling of communication protocols by graph transformation. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 348–363. Springer, Heidelberg (2006)
13. Rensink, A., Distefano, D.: Abstract graph transformation. In: International Workshop on Software Verification and Validation (SVV) (2005)
14. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformations, Foundations, vol. 1. World Scientific, Singapore (1997)
15. Varró, D.: Automated formal verification of visual modeling languages by model checking. Journal of Software and Systems Modelling (2003)
16. Zave, P.: Ideal connection paths in DFC. Technical report, AT&T Research (November 2003)
17. Zave, P.: Address translation in telecommunication features. ACM Trans. Softw. Eng. Methodol. 13(1), 1–36 (2004)
18. Zave, P.: Requirements for routing in the application layer. In: Murphy, A.L., Vitek, J. (eds.) COORDINATION 2007. LNCS, vol. 4467, pp. 19–36. Springer, Heidelberg (2007)

Formal Probabilistic Analysis of Stuck-at Faults in Reconfigurable Memory Arrays

Osman Hasan, Naeem Abbasi, and Sofiène Tahar

Dept. of Electrical & Computer Engineering, Concordia University
1455 de Maisonneuve W., Montreal, Quebec, H3G 1M8, Canada
{o_hasan,n_ab,tahar}@ece.concordia.ca

Abstract. Reconfigurable memory arrays with spare rows and columns are quite frequently used as reliable data storage components in present age System-on-Chips (SoCs). The spare memory rows and columns can be utilized to automatically replace rows or columns that are found to contain a cell fault after fabrication. One of the biggest SoC design challenges is to estimate, prior to the actual fabrication process, the right number of these spare rows and spare columns for meeting the reliability specifications. Traditionally, computer simulation techniques are used to perform probabilistic analysis of reconfigurable memory arrays but they provide inaccurate results. To ensure accurate analysis and thus more reliable SoC designs, we propose, in this paper, a probabilistic theorem proving approach in the domain of reconfigurable memory array analysis. We present a higher-order-logic stuck-at fault model for reconfigurable memory arrays, based on which, we illustrate the formal verification of some key statistical properties related to the number of stuck-at faults and the repairability condition.

1 Introduction

Embedded memory is the most dominant component in terms of silicon area of any System-on-Chip (SoC) these days. Applications such as mobile communication devices, medical and industrial signal processing and digital switching systems used in computer networks all require large amounts of memory. It is expected that by the end of this decade about 90% of the chip area on a typical SoC will be taken up by memory of one type or the other (Static, Dynamic, Flash, or Content addressable) [15]. These memories on a chip are usually organized in very highly optimized structures in an effort to reduce cost. Extremely small memory cell sizes and the fact that a significant amount of the chip area is taken by compact memory arrays, makes memories more prone to defects during fabrication than regular logic. The defects in a memory can render the whole SoC useless. Even in mature fabrication processes where the defect densities tend to be small, the throwing away of any chip is considered unacceptable because of its adverse effect on yield. Moreover, these defects may also lead to devastating situations when the bug is not caught in the testing phase and the faulty chip is used in a safety critical domain, such as medicine, military and transportation.

In order to analyze the effects of memory defects, memory fault models are constructed that describe how a fault in memory might occur and predict the resulting device behavior. There are four main types of faults that may occur in a memory array: stuck-at faults, transition faults, coupling faults, and neighborhood pattern sensitive faults [20]. Stuck-at faults, which occur when a memory cell never changes its state, i.e., it is always stuck in one state, is the most commonly used fault model for analyzing memory arrays and logic. The information gathered from the fault models is utilized for the development of techniques for detecting and repairing memory faults. One such widely used technique is to add some redundancy to the memory array during the design phase. This way even after fabrication, we can repair some of the memory faults by replacing the rows or columns of the memory array containing faulty memory cells with the available spare rows or columns. Memories fabricated with these spare rows and columns are usually termed as *reconfigurable memory arrays*. This technique poses an interesting solution to the memory faults problem but comes with a bigger design challenge of estimating the right number of spare rows and columns for meeting reliability specifications. If a combination of spare rows and columns exists such that all faults from the memory array can be eliminated then such a combination of spare rows and columns is called a *repair solution*, and the array is called *repairable*. The repairability problem of a reconfigurable memory array is similar to the vertex cover problem of the bipartite graph and is known to be an NP complete problem [16]. Thus, probabilistic analysis and some graph theory principles are usually utilized to obtain reasonable solutions [25,18,2].

Today, simulation is the most commonly used computer based probabilistic analysis tool for reconfigurable memory arrays, e.g., see [21,23]. Most simulation based memory array analysis software provide a programming environment for defining functions that approximate random variables for probability distributions. The random elements, such as fault occurrences, in a given memory array are modeled by these functions and the model is analyzed using computer simulation techniques [5], such as the Monte Carlo method [19], where the main idea is to approximately answer a query on a probability distribution by analyzing a large number of samples. Statistical quantities, such as expectation and variance, may then be calculated, based on the data collected during the sampling process, using their mathematical relations in a computer. Due to the inherent nature of simulation coupled with the usage of computer arithmetic, the probabilistic analysis results attained by the simulation approach can never be termed as 100% accurate. Moreover, simulation requires an enormous amount of CPU time for attaining meaningful estimates. We generally need to acquire hundreds of thousands of samples to estimate the desired probabilistic quantities and this fact makes the simulation approach impractical when each sample acquisition step involves extensive computations, which is usually the case for analyzing reconfigurable memory arrays due to their large capacities. Thus, simulation should not be relied upon for the analysis of reconfigurable memory arrays, especially when they are used in safety critical areas, where inaccuracies and inadequacies in the analysis may even result in the loss of human lives.

In the past couple of decades, formal methods [8] have been successfully used for the precise analysis of a verity of hardware and software systems. The rigorous exercise of developing a mathematical model for the given system and analyzing this model using mathematical reasoning usually increases the chances for catching subtle but critical design errors that are often ignored by traditional techniques like simulation. Given the sophistication of the present age memory reconfigurable arrays and their extensive usage in SoCs for safety critical applications, there is a dire need of using formal methods in this domain. However, to the best of our knowledge, due to the random and unpredictable occurrence pattern of memory array faults, the usage of formal methods for their analysis has never been attempted. Some of the major reasons for this include the inability to precisely reason about statistical properties, such as expectation and variance, in the case of state-based approaches and the fear of huge proof efforts involved in modeling and reasoning about random occurrence patterns of memory faults in the case of theorem proving with expressive logics.

We believe that due to the recent developments in the formalization of probability theory concepts [14,11,12,13], we are now at the stage where we can handle the probabilistic analysis of reconfigurable memory arrays in a higher-order-logic theorem prover [6] with reasonable amount of modeling and verification efforts. We illustrate the practical effectiveness of this argument by presenting the higher-order-logic theorem proving based analysis of the repairability problem for stuck-at faults in this paper. Even though, we concentrate on stuck-at faults here, the presented approach is quite general and can be essentially utilized to conduct the analysis of other kinds of memory faults as well.

This paper presents a three step approach for tackling the repairability problem. We proceed by formally expressing a stuck-at fault model for reconfigurable memory arrays in higher-order logic. Our formalization utilizes precise random variable functions to express the random components in the model. Secondly, we utilize our formal model to express and verify statistical properties, such as expectation and variance of the number of faults in terms of memory array and spare rows and columns sizes, as higher-order logic theorems. Finally, this formal statistical information is utilized to formally verify a relation that ascertains that a large square memory array is almost always repairable (with probability 1) if stuck-at faults are independent and identically distributed with a specific probability. This result can now be used to accurately estimate the number of spare rows and columns required for reliable operation against stuck-at faults of any reconfigurable memory array without any CPU time constraints. We have utilized the higher-order-logic theorem prover HOL [7] for this work. The main motivation behind using the HOL theorem prover is the fact that it contains most of the foundational probability theory work that we build upon.

The rest of the paper is organized as follows: Section 2 presents some related work. Section 3 provides an overview of HOL probabilistic analysis related foundations that we build upon to conduct the analysis of reconfigurable memory arrays in this paper. In Section 4, we present our formal probabilistic model of the number of stuck-at faults in memory arrays. This is followed by the formal verification of some statistical properties and the repairability condition in Section 5. Finally, Section 6 concludes the paper.

2 Related Work

Simulation techniques are very commonly used in the yield and repairability analysis for memory arrays. One such yield analysis tool , described in [23], for integrated circuits containing multiple, possibly different, repairable embedded memories. Pseudo random faults are generated based on memory area, defect density, and fault distribution. Then, using a flexible array model, optimal numbers of spare rows and columns for a given memory are determined. The tool is also used to determine the effectiveness of various repair algorithms. In [21] a Built-in self repair (BISR) technique is presented that merges error correction coding schemes and self repair using spare rows and columns. The technique is validated through simulation and it is shown that for defect densities as high as 10^{-2} % (or when 3% of cells are defective) near 100% memory yield can be achieved and thus is suitable for nanometer CMOS process generations.

When memory sizes become large, analysis through simulation very quickly becomes computationally difficult to handle. Paper-and-pencil based analytical analysis have been traditionally used for such cases. A memory array probability model represents either the occurrence of individual faults or the total number of faults as a random variable and thus allows reasoning about statistical properties. Questions, such as “given a certain fault distribution and number of faults can almost every memory array be repaired”, or “with how many faults a memory array can almost never be repaired”, can then be answered [2,25,18].

To the best of our knowledge, higher-order-logic theorem proving has never been used for the probabilistic analysis of any memory reconfigurable array so far. Though, some useful research related to the foundations of probabilistic analysis is available in the open literature. Random variables can be formalized and verified, based on their probability distribution properties, using the methodology proposed in [14]. In fact, [14] presents the formalization of some discrete random variables along with their verification, based on the corresponding PMF properties. Building upon Hurd’s formalization framework [14], the sampling algorithms of a few continuous random variables have also been formalized and verified [11]. In [12,10], we extended Hurd’s formalization framework with a formal definition of expectation. This definition is then utilized to formalize and verify the expectation and variance characteristics associated with discrete random variables that attain values in *natural* numbers only.

Besides theorem proving, another formal method that can be used for conducting precise probabilistic analysis of reconfigurable memory arrays is probabilistic model checking [1,22]. The main idea behind this approach is to construct a precise state-based mathematical model of the given memory array and then utilize this model to exhaustively verify the intended, formally represented, probabilistic properties, such as the probability of number of faults being less than some threshold value in a given memory array. Besides the accuracy of the results, the most promising feature of probabilistic model checking is the ability to perform the analysis automatically. On the other hand, it is limited to systems that can only be expressed as probabilistic finite state machines or Markov chains. Another major limitation of the probabilistic model checking approach is state space explosion [3], due to which large capacity memories cannot be analyzed using this approach. Similarly, to the best of our knowledge, it has not been possible to precisely reason about statistical quantities, such as variance and tail distribution bounds, using probabilistic model checking so far. The most that has been reported in this domain is the evaluation of a small subset of expected values in a couple of model checkers, such as PRISM [17] and VESTA [24]. Because of the above mentioned limitations, probabilistic model checking is not feasible for analyzing memory array repairability problem as the models are usually large and most of the decision making in this domain is made based on statistical quantities. Whereas, the proposed higher-order-logic theorem proving based approach, allows us to analyze a wider range of memory arrays without any modeling limitations, such as the restrictiveness to Markovian models or the state-space explosion problem, and formally verify statistical properties, as will be seen in the next section.

3 Probabilistic Analysis in HOL

The foremost criteria for implementing a theorem proving based probabilistic analysis framework is to be able to formalize and verify random variables in higher-order logic. Hurd's PhD thesis [14] can be considered a pioneering work in this regard as it presents a methodology for the formalization and verification of probabilistic algorithms in the HOL theorem prover. Random variables can be formalized in higher-order logic as deterministic functions with access to an infinite Boolean sequence $\mathbb{B}^\infty$; a source of infinite random bits [14]. These deterministic functions make random choices based on the result of popping the top most bit in the infinite Boolean sequence and may pop as many random bits as they need for their computation. When the functions terminate, they return the result along with the remaining portion of the infinite Boolean sequence to be used by other programs. Thus, a random variable which takes a parameter of type α and ranges over values of type β can be represented in HOL by the function.

$$\mathcal{F} : \alpha \rightarrow B^\infty \rightarrow \beta \times B^\infty$$

As an example, consider the Bernoulli($\frac{1}{2}$) random variable that returns 1 or 0 with equal probability $\frac{1}{2}$. It can be formalized in HOL as follows

```
⊢ bit = (λs. if shd s then 1 else 0, stl s)
```

where s is the infinite Boolean sequence and `shd` and `stl` are the sequence equivalents of the list operation *'head'* and *'tail'*. The probabilistic programs can also be expressed in the more general state-transforming monad where the states are the infinite Boolean sequences.

```
⊢ ∀ a s. unit a s = (a,s)
⊢ ∀ f g s. bind f g s = g (fst (f s)) (snd (f s))
```

The `unit` operator is used to lift values to the monad, and the `bind` is the monadic analogue of function application. All monad laws hold for this definition, and the notation allows us to write functions without explicitly mentioning the sequence that is passed around, e.g., function *bit* can be defined as

```
⊢ bit_monad = bind sdest (λb. if b then unit 1 else unit 0)
```

where `sdest` gives the head and tail of a sequence as a pair (*shd* `s`, *stl* `s`).

[14] also presents some formalization of the mathematical measure theory in HOL, which can be used to define a probability function $\mathbb{P}$ from sets of infinite Boolean sequences to *real* numbers between 0 and 1. The domain of $\mathbb{P}$ is the set $\mathcal{E}$ of events of the probability space. Both $\mathbb{P}$ and $\mathcal{E}$ are defined using the Carathéodory's Extension theorem, which ensures that $\mathcal{E}$ is a σ-algebra: closed under complements and countable unions. The formalized $\mathbb{P}$ and $\mathcal{E}$ can be used to formally verify probabilistic properties, e.g.,

$$\vdash \mathbb{P}\ \{\mathtt{s}\ |\ \mathtt{fst\ (bit\ s)\ =\ 1}\} = \frac{1}{2}$$

where the HOL function `fst` selects the first component of a pair and $\{x|C(x)\}$ represents a set of all x that satisfy the condition C. The above approach has been successfully used to formalize and verify both discrete [14,13] and continuous random variables [11] in HOL.

Expectation theory plays a vital role in the domain of probabilistic analysis as it is a lot easier to judge performance issues based on the average value of a random variable, which is a single number, rather than its distribution function. Building on the above mentioned probabilistic analysis infrastructure, [12] presents a higher-order-logic definition of expectation for discrete random variables. This function has been used to successfully verify the average values of most of the commonly used discrete random variables. For example, [13] presents the verification of average value of the Binomial random variable, which will be later utilized in this paper for memory array analysis.

Lemma 1: *Expectation of Binomial(m,p) Random Variable*

```
⊢ ∀ m p. 0 ≤ p ∧ p ≤ 1 ⇒ expec (λs. prob_bino m p s) = m p
```

where (λ`x.t`) represents a lambda abstraction function in HOL that maps its argument x to $t(x)$ and `prob_bino` is the HOL function for the Binomial random variable modeled using the above mentioned approach.

The higher-order-logic probabilistic analysis approach was further strengthened by some additional formalization related to expectation theory in [10]. This includes a formal definition of the variance characteristic, which is used for measuring dispersion of a random variable. This definition of variance can be utilized to verify variance characteristics of most of the commonly used discrete random variables, e.g., [13] presents the verification of variance of the Binomial random variable as the following theorem.

Lemma 2: *Variance of Binomial(m,p) Random Variable*

$\vdash \forall$ `m p. 0` $\leq$ `p` $\wedge$ `p` $\leq$ `1`
  $\Rightarrow$ `variance (`λ`s. prob_bino m p s) = m p (1 - p)`

The work in [13], also includes the verification of some classical properties of expectation and variance in HOL. One such property is the Chebyshev's inequality, which plays a vital role in verifying tail distribution bounds of probabilistic systems within the HOL theorem prover and is given below

Lemma 3: *Chebyshev's Inequality*

$\vdash \forall$ `R a. (0` $<$ `a)` $\wedge$ `(0` $<$ `variance R)` $\wedge$
  `(summable(`λ`n. n` $\mathbb{P}$`{s | fst (R s) = n}))` $\wedge$
  `(summable(`λ`n.` n^2 $\mathbb{P}$`{s | fst (R s) = n}))`
$\Rightarrow$ $\mathbb{P}$ `{s | abs (fst (R s) - expec R)` $\geq$ `a}` $\leq \frac{\text{variance R}}{\text{a}^2}$

where the HOL predicate `summable` is *True* if the infinite summation of its *real* sequence argument exists [9], i.e., $\exists x. \lim_{k\rightarrow\infty} \sum_{n=0}^{k} f(n) = x$. Thus, the *summable* assumptions in the above theorem state that the theorem is only valid for a random variable R with well-defined expectation and variance values.

In this paper, we utilize the above mentioned infrastructure for conducting formal probabilistic analysis of reconfigurable memory arrays, a novelty that to the best of our knowledge does not exist in the open literature so far.

4 Formal Stuck-at Fault Memory Model

In this section, we develop a formal generic stuck-at fault model for reconfigurable memory arrays. This model will be used to formally reason about the statistical properties and repairability of the memory arrays in the next section. Our formalization approach is mainly inspired by the analytical model developed in [25] for the paper-and-pencil based analysis of reconfigurable memory arrays.

The reconfigurable memory array can be modeled as a bipartite graph (R, C, F). In this bipartite graph, R represents the set of nodes representing rows of the memory array, C is the set of nodes representing the columns of memory array, and F is a set of edges, with each edge connecting one node in the set R to a node in the set C, and represents a fault in the memory array. It is important to note here that the number of elements in the set F and their identities is a random quantity as fault occurrence is an unpredictable event. Therefore, we associate a probability p with every possible pair combination of the elements of the sets R and C of being included in the set F. Also, the occurrence of stuck-at faults, and thus the inclusion of a pair in the set F, is assumed to be independent and identically distributed in this model.

For illustration purposes, consider a square memory array of size n x n with *sr* spare rows and *sc* spare columns and four stuck-at faults, as shown in Figure 1.(a). The corresponding bipartite graph model of the memory array is given in Figure 1.(b). In this model, each of the four faults is represented as an edge connecting a row and column node.

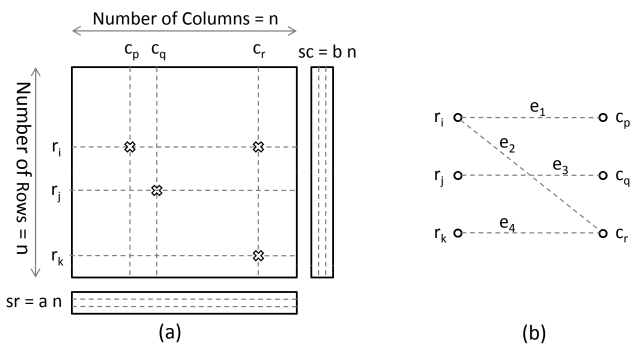


Fig. 1. Memory Array Model

A stuck-at fault occurring at location (x, y) in the memory array can be repaired by replacing either row x or column y with a spare row or a spare column. Thus, in the worst case scenario when we require one row or column to repair a single fault only, a memory array is considered to be absolutely repairable if its total number of stuck-at faults is less than the available number of spare columns or rows. This repairability problem is similar to the vertex cover problem of the bipartite graph and is known to be an NP complete problem [16]. Therefore, we consider solutions to this problem using probability theory and define the probability of repairability, using our memory array model, as follows

$$\Pr(|F| \leq sr + sc) \tag{1}$$

where Pr and $|F|$ represent the probability function and cardinality of a set F, respectively. Equation (1) represents the probability of the event when the number of stuck-at faults $|F|$, a random quantity, is less than the total number of spare rows and columns $sr + sc$. We can express Equation (1) in terms of the number of rows or columns of a square n x n reconfigurable memory array as

$$\Pr(|F| \leq (a + b)n) \tag{2}$$

where $a = \frac{sr}{n}$ and $b = \frac{sc}{n}$. The values of a and b are bounded in the real interval $[0, 1]$, since the number of spare rows and spare columns is usually a small fraction of the total number of rows and columns in the array and can never exceed it.

In this paper, our primary goal is to formally verify that if the probability of stuck-at fault occurrence is given by the following expression

$$p = \frac{(a + b)}{n} - \frac{w(n)}{n\sqrt{n}} \tag{3}$$

then the memory array is almost always repairable, whereas $w(n) \to \infty$ as $n \to \infty$. The term almost always repairable in the above context means that the probability of repairability tends to 1 as n becomes very very large. The above expression for the stuck-at fault occurrence probability has been initially proposed and analyzed using informal techniques in [25]. Our contribution in this paper is to formally verify the above argument using the HOL theorem prover.

We proceed in this direction by first modeling the number of faults or the cardinality of the set F using the following higher-order-logic functions.

Definition 1: *Stuck-at Fault Memory Model*

```
⊢ (∀ p. mem_fault_model_helper 0 p = unit 0) ∧
   ∀ c p. mem_fault_model_helper (c + 1) p =
     bind (mem_fault_model_helper c p)
  (λa. bind (prob_bern p) (λb. unit (if b then (a+1) else a)))

⊢ (∀ c p. mem_fault_model 0 c p = unit 0) ∧
   ∀ r c p. mem_fault_model (r + 1) c p =
     bind (mem_fault_model r c p)
   (λa. bind (mem_fault_model_helper c p) (λb. unit (a + b)))
```

The function `mem_fault_model` accepts three parameters: the cardinalities of the sets R and C and the probability of fault occurrence p. It recursively manipulates these three parameters, with the help of the function `mem_fault_model_helper` and returns the number of faults found in the memory array of size $|R|$ x $|C|$. It is important to note that the fault occurrence behavior, which is the random component in this model, is represented by the formalized Bernoulli random variable function `prob_bern` [14] above. The function `mem_fault_model` basically performs a Bernoulli trail, with the probability of obtaining a *True* being equal to the probability of fault occurrence, for each cell of the memory array and returns the total number of *True* outcomes obtained.

Now, in order to verify the condition of repairability, given in Equation (3), we define the following special case of our general memory model.

Definition 2: *Stuck-at Fault Memory Model for Repairability Problem*

⊢ ∀ n a b w. `mem_fault_model_rep n a b w =`
  `mem_fault_model n n` $(\frac{(a+b)}{n} - \frac{w(n)}{n\sqrt{n}})$

The function `mem_fault_model_rep` accepts four parameters: the cardinality of the sets R and C of a square reconfigurable memory array as a *natural* number n, the fractions of spare rows and columns as *real* numbers a and b, respectively, and the *real* sequence w with data type (*natural* $\rightarrow$ *real*). It utilizes the function `mem_fault_model`, given in Definition 1, to return the number of stuck-at faults for the specific case of a square n x n memory array with the fault occurrence probability equal to the expression, given in Equation (3).

For simplifying the interactive proofs related to the function `mem_fault_model_rep`, it can be alternately expressed as follows

Lemma 4: *Alternate Stuck-at Fault Memory Model for Repairability Problem*

⊢ ∀ n a b w.
  `mem_fault_model_rep n a b w = prob_bino` n^2 $(\frac{(a+b)}{n} - \frac{w(n)}{n\sqrt{n}})$

The proof of the above lemma is primarily based on the fact that the stuck-at fault occurrences in our memory array model are independent and identically distributed and so are the Bernoulli random variables used in the function definition of `mem_fault_model_rep`. This allows us to express the summation of n^2 Bernoulli(p) random variables in the function `mem_fault_model_rep` as a Binomial(n^2, p) random variable, since Binomial(m, p) random variable basically counts the number of successes in m independent and identically distributed Bernoulli trials, with a success probability p [4].

5 Statistical Properties and Repairability Condition

In this section, we utilize the function `mem_fault_model_rep` to formally verify a couple of statistical properties regarding the number of faults and the almost always repairability condition for an n x n reconfigurable memory array with stuck-at fault occurrence probability given by Equation (3). These verification results play a vital role in designing reliable reconfigurable memory arrays.

5.1 Average Number of Stuck-at Faults

With the probability of stuck-fault occurrence, given by Equation (3), the average number of stuck-at faults for an n x n memory array is given by

$$Ex[|F|] = n^2\left(\frac{(a+b)}{n} - \frac{w(n)}{n\sqrt{n}}\right) \tag{4}$$

This property can be formally expressed in higher-order logic using our formal definition of the number of faults, given in Definition 2, as follows.

Theorem 1: *Average Number of Stuck-at Faults*

$$\begin{array}{l} \vdash \forall\ \mathtt{a\ b\ n\ w.} \\ \quad (\mathtt{0} \le \mathtt{a}) \wedge (\mathtt{a} \le \mathtt{1}) \wedge (\mathtt{0} \le \mathtt{b}) \wedge (\mathtt{b} \le \mathtt{1}) \wedge (\mathtt{1} < \mathtt{n}) \wedge \\ \quad (\forall\ \mathtt{n.}\ (\mathtt{0} < \mathtt{w(n)}) \wedge (\mathtt{w(n)} < (\mathtt{a}+\mathtt{b})\sqrt{\mathtt{n}})) \\ \Rightarrow \mathtt{expec}\ (\lambda \mathtt{s.\ mem\_fault\_model\_rep\ n\ a\ b\ w\ s}) = \mathtt{n}^2\left(\frac{(\mathtt{a}+\mathtt{b})}{\mathtt{n}} - \frac{\mathtt{w(n)}}{\mathtt{n}\sqrt{\mathtt{n}}}\right) \end{array}$$

The first four assumptions in the above theorem ensure that the fractions a and b are bounded by the interval $[0, 1]$ as described in the previous section. Whereas, the precondition $1 < n$ has been used in order to ensure that the given memory array has more than one cell. The last assumption is about the real sequence w and basically provides its upper and lower bounds. These bounds have been used in order to prevent the stuck-at fault occurrence probability p, given in Equation (3), from falling outside its allowed interval $[0, 1]$. It is interesting to note that no such restriction on the sequence w was imposed in the paper-and-pencil based analysis of the repairability problem given in [25]. This fact clearly demonstrates the strength of formal methods based analysis as it allowed us to highlight this corner case, which if ignored could lead to the invalidation of the whole repairability analysis. The conclusion of Theorem 1 presents the mathematical relation given in Equation (4).

The HOL proof for Theorem 1 is based on Lemma 4 and the expectation relation for the Binomial random variable, given in Lemma 1. The proof involves some arithmetic reasoning to verify that the probability p, given in Equation (3), lies in the interval $[0, 1]$, which is a precondition for Lemma 1.

5.2 Variance of the Number of Stuck-at Faults

The variance of the number of stuck-at faults for an n x n memory array, with the probability of stuck-at fault occurrence given by Equation (3), is given by

$$Var[|F|] = n^2(\frac{(a+b)}{n} - \frac{w(n)}{n\sqrt{n}})(1 - (\frac{(a+b)}{n} - \frac{w(n)}{n\sqrt{n}})) \tag{5}$$

This property can be formally expressed in HOL as follows

Theorem 2: *Variance of the Number of Stuck-at Faults*

$\vdash \forall$ a b n w s.
$(0 \leq a) \wedge (a \leq 1) \wedge (0 \leq b) \wedge (b \leq 1) \wedge (1 < n) \wedge$
$(\forall$ n. $(0 < w(n)) \wedge (w(n) < (a+b)\sqrt{n}))$
$\Rightarrow$ variance
(λs. mem_fault_model_rep n a b w s) =
$n^2(\frac{(a+b)}{n} - \frac{w(n)}{n\sqrt{n}})(1 - (\frac{(a+b)}{n} - \frac{w(n)}{n\sqrt{n}}))$

and verified using Lemma 2 just like Lemma 1 was used to verify Theorem 1.

5.3 Tail Distribution Bound for the Number of Stuck-at Faults

A tail distribution bound of the number of stuck-at faults for our n x n memory array, with the probability of stuck-at fault occurrence, given by Equation (3), can be expressed as follows.

$$Pr(|F| \leq (a+b)n) \geq 1 - \frac{n^2(\frac{(a+b)}{n} - \frac{w(n)}{n\sqrt{n}})(1 - (\frac{(a+b)}{n} - \frac{w(n)}{n\sqrt{n}}))}{n(w(n))^2} \tag{6}$$

Whereas, the corresponding HOL theorem is as follows.

Theorem 3: *Tail Distribution Bound for the number of Stuck-at Faults*

$\vdash \forall$ a b n w s.
$(0 \leq a) \wedge (a \leq 1) \wedge (0 \leq b) \wedge (b \leq 1) \wedge (1 < n) \wedge$
$(\forall$ n. $(0 < w(n)) \wedge (w(n) < (a+b)\sqrt{n}))$
$\Rightarrow (\mathbb{P}$ {s | (fst (mem_fault_model_rep n a b w s)) $\leq$ (a + b)n} $\geq$
1 - $(\frac{n^2(\frac{(a+b)}{n} - \frac{w(n)}{n\sqrt{n}})(1-(\frac{(a+b)}{n} - \frac{w(n)}{n\sqrt{n}}))}{n(w(n))^2})$

We proceed with the verification of this theorem by splitting its proof goal into two subgoals using the less-than-or-equal-to transitive property as follows.

$$\mathbb{P}\ \{\texttt{s | (fst (prob\_bino n}^2\ (\tfrac{(a+b)}{n} - \tfrac{w(n)}{n\sqrt{n}})\ \texttt{s)} > (a+b)n - 2\sqrt{n}w(n))\ \wedge$$
$$\texttt{(fst (prob\_bino n}^2\ (\tfrac{(a+b)}{n} - \tfrac{w(n)}{n\sqrt{n}})\ \texttt{s)} < (a+b)n\ \}$$
$$\leq \mathbb{P}\ \{\texttt{s | (fst (prob\_bino n}^2\ (\tfrac{(a+b)}{n} - \tfrac{w(n)}{n\sqrt{n}})\ \texttt{s))} \leq \texttt{(a + b)n}\}$$

$$1 - \frac{n^2(\frac{(a+b)}{n} - \frac{w(n)}{n\sqrt{n}})(1-(\frac{(a+b)}{n} - \frac{w(n)}{n\sqrt{n}}))}{(nw(n)w(n))} \leq$$
$$\mathbb{P}\ \{\texttt{s | (fst (prob\_bino n}^2\ (\tfrac{(a+b)}{n} - \tfrac{w(n)}{n\sqrt{n}})\ \texttt{s))} > (a+b)n - 2\sqrt{n}w(n)\ \wedge$$
$$\texttt{(fst (prob\_bino n}^2\ (\tfrac{(a+b)}{n} - \tfrac{w(n)}{n\sqrt{n}})\ \texttt{s))} < (a+b)n\ \}$$

The first subgoal can be verified using the basic probability axiom ($\forall A\ B.A \subseteq B \Rightarrow (Pr(A) \leq Pr(B))$) since the set on the left-hand-side (LHS) of the inequality is a subset of the set on the right-hand-side (RHS). Whereas, by rewriting the two inequalities in the argument of the probability function of subgoal 2 using absolute value theorem $((|y-x| < d) = (x - d < y < x + d)$ we get:

$$1 - \frac{n^2(\frac{(a+b)}{n} - \frac{w(n)}{n\sqrt{n}})(1-(\frac{(a+b)}{n} - \frac{w(n)}{n\sqrt{n}}))}{(nw(n)w(n))} \leq$$
$$\mathbb{P}\ \{\ \texttt{s | |fst (prob\_bino n}^2\ (\tfrac{(a+b)}{n} - \tfrac{w(n)}{n\sqrt{n}})\ \texttt{s) - ((}a+b)n - \sqrt{n}w(n))|$$
$$< \sqrt{n}w(n)\ \}$$

Now using the complement probability law ($\forall A.Pr(\bar{A}) = 1 - Pr(A)$) along with Theorems 1 and 2, we can rewrite the above sub goal as follows

$$\mathbb{P}\ \{\texttt{s | |fst (prob\_bino n}^2\ (\tfrac{(a+b)}{n} - \tfrac{w(n)}{n\sqrt{n}})\ \texttt{s) -}$$
$$\texttt{expec (}\lambda\texttt{s. prob\_bino n}^2\ (\tfrac{(a+b)}{n} - \tfrac{w(n)}{n\sqrt{n}})\ \texttt{s)|} \geq \sqrt{n}w(n)\ \} \leq$$
$$\frac{\texttt{variance(}\lambda\texttt{s.prob\_bino n}^2(\frac{(a+b)}{n} - \frac{w(n)}{n\sqrt{n}})\texttt{s)}}{(\sqrt{n}w(n))^2}$$

The above subgoal can now be discharged from the HOL goal stack by using Chebyshev's inequality, given in Lemma 3, along with some arithmetic reasoning.

5.4 Repairability Problem

Now, we use the statistical properties verified so far to analyze the repairability problem, i.e., an n x n reconfigurable memory array with the probability of stuck-at fault occurrence given by Equation (3), is almost always repairable.

$$\lim_{n\to\infty} \Pr(|F| \leq (a+b)n) = 1 \tag{7}$$

The corresponding HOL theorem is as follows

Theorem 4: *Repairability Problem of Stuck-at Faults*

$$\vdash \forall\ \texttt{a b w.}\ (0 \leq a) \wedge (a \leq 1) \wedge (0 \leq b) \wedge (b \leq 1) \wedge$$
$$(\forall\ n.\ (0 < w(n)) \wedge (w(n) < (a+b)\sqrt{n})) \wedge$$
$$(\texttt{lim}\ (\lambda n.\ \tfrac{1}{w(n)}) = 0)$$
$$\Rightarrow (\texttt{lim}\ (\lambda n.$$
$$\mathbb{P}\ \{\texttt{s | (fst (num\_of\_faults n a b w s))} \leq \texttt{(a + b) n}\ \}) = 1))$$

where `lim M` represents the HOL formalization of the limit of a real sequence M (i.e., $lim\ M = \lim_{n\to\infty} M(n)$) [9]. The new assumption $(\mathtt{lim}(\lambda \mathtt{n}.\frac{1}{\mathtt{w(n)}}) = 0)$ formally represents the intrinsic characteristic of *real* sequence w that it tends to infinity as its *natural* argument becomes very very large.

We proceed with the verification of Theorem 4 by first splitting its proof goal into the following two subgoals, based on some simple arithmetic reasoning.

$$\mathtt{lim}(\lambda \mathtt{n}.\ \mathbb{P}\{\mathtt{s}\ |\ \mathtt{fst\ (mem\_fault\_model\_rep\ n\ a\ b\ w\ s)} \leq \mathtt{(a + b)n}\}) \leq 1$$

$$1 \leq \mathtt{lim}(\lambda \mathtt{n}.\ \mathbb{P}\{\mathtt{s}\ |\ \mathtt{fst\ (mem\_fault\_model\_rep\ n\ a\ b\ w\ s)} \leq \mathtt{(a + b)n}\})$$

The first subgoal can be verified using the basic probability axiom ($\forall A.Pr(A)$ ≤ 1). Whereas, we utilize Theorem 3 and the transitivity property of less-than-or-equal-to for real numbers to rewrite the second subgoal as follows.

$$1 \leq \mathtt{lim}\ (\lambda \mathtt{n}.\ 1 - \frac{\mathtt{n}^2(\frac{\mathtt{a+b}}{\mathtt{n}} - \frac{\mathtt{w(n)}}{\mathtt{n}\sqrt{\mathtt{n}}})(1 - \frac{\mathtt{a+b}}{\mathtt{n}} + \frac{\mathtt{w(n)}}{\mathtt{n}\sqrt{\mathtt{n}}})}{\mathtt{n(w(n))}^2})$$

The expression in the RHS of the above inequality can be rewritten as follows using some arithmetic reasoning.

$$1 \leq \mathtt{lim}\ (\lambda \mathtt{n}.\ 1 - ((\frac{\mathtt{a+b}}{\mathtt{w(n)}} - \frac{1}{\sqrt{\mathtt{n}}})(\frac{1}{\mathtt{w(n)}} - \frac{\mathtt{a+b}}{\mathtt{nw(n)}} + \frac{1}{\mathtt{n}\sqrt{\mathtt{n}}})))$$

This subgoal can now be verified as the limit value of the expression on the RHS tends to 1, since all the denominator terms in this expression tend to ∞ as n becomes very very large. This also concludes the proof for Theorem 4.

Our results clearly demonstrate the effectiveness of the theorem proving based reconfigurable memory array analysis approach. Due to the formal nature of the model and inherent soundness of theorem proving, we have been able to verify the properties of interest regarding the given memory array with 100% precision; a novelty which is not available in simulation. Similarly, due to the high expressibility of higher-order logic we have been able to formally reason about statistical properties of the problem that cannot be analyzed using a probabilistic model checker. The proposed approach is also superior than the paper-and-pencil proof methods in terms of accuracy. In the paper-and-pencil approach, the proof checking and associated bookkeeping is an error prone process, specially when dealing with large proofs, and thus often leads either to wasted time and effort or a wrong result. On the other hand, in theorem proving, these complicated tasks are done by the computer within a sound core, which is based on a very few axioms and inference rules. Each proven theorem can be logically traced back to these basic axioms and the associated proof steps can be linked to the basic inference rules. Due to this inherent soundness, it is impossible to prove wrong statements in a theorem prover.

The above mentioned additional benefits, associated with the theorem proving approach, are attained at the cost of the time and effort spent, while formalizing the memory array and formally reasoning about its properties, by the user. But, the fact that we were building on top of already verified probability theory related results helped significantly in this regard as this analysis only consumed approximately 80 man-hours and 1200 lines of HOL code by an expert user.

6 Conclusions

In this paper, we utilized the mathematical probability theory formalized in a higher-order-logic theorem prover to analyze reconfigurable memory arrays in the presence of stuck-at faults. To the best of our knowledge, this is the first study on using these kind of techniques for such an application. We developed a higher-order-logic based formal stuck-at fault model for reconfigurable memory arrays, and based on this model we formally verified some key statistical properties and repairability condition. The rigorous exercise of developing a computer based formal model for the memory array and analyzing it using mechanized mathematical reasoning allowed us to discover a couple of critical assumptions that are missed by almost all of the paper-and-pencil based analysis, that we came across, of a similar problem. Due to the formal nature of the models and the inherent soundness of theorem proving systems, the analysis is guaranteed to provide exact answers. These feature makes the proposed approach very useful for the probabilistic analysis of memory arrays that are to be used in safety critical and highly sensitive areas.

Our approach for the probabilistic analysis of stuck-at faults in memory reconfigurable arrays is quite general and can be extended and easily adapted to conduct precise probabilistic analysis of other kinds of fault models, like transition faults, coupling faults, and neighborhood pattern sensitive faults, as well. The random or unpredictable elements found in these models can be represented using an appropriate random variable from the existing library of formalized discrete [14,12,13] and continuous random variables [11], and the precise statistical quantities associated with the parameters of interest may then be verified within the sound core of a higher-order-logic theorem prover. For example, the probabilistic analysis approach for coupling faults [25] can be adapted in a theorem prove using the formal definition of the Binomial random variable along with the theorems regarding its expectation and variance. Similarly, other statistical properties, such as the conditions for irrepairability and tail distribution bounds based on Markov's inequality, can also be verified.

References

1. Baier, C., Haverkort, B., Hermanns, H., Katoen, J.P.: Model Checking Algorithms for Continuous time Markov Chains. IEEE Transactions on Software Engineering 29(4), 524–541 (2003)
2. Blough, D.M.: Performance Evaluation of a Reconfiguration-Algorithm for Memory Arrays containing Clustered Faults. IEEE Transactions on Reliability 45(2), 274–284 (1996)
3. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (2000)
4. DeGroot, M.: Probability and Statistics. Addison-Wesley, Reading (1989)
5. Devroye, L.: Non-Uniform Random Variate Generation. Springer, Heidelberg (1986)
6. Gordon, M.J.C.: Mechanizing Programming Logics in Higher-0rder Logic. In: Current Trends in Hardware Verification and Automated Theorem Proving, pp. 387–439. Springer, Heidelberg (1989)

7. Gordon, M.J.C., Melham, T.F.: Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic. Cambridge University Press, Cambridge (1993)
8. Gupta, A.: Formal Hardware Verification Methods: A Survey. Formal Methods in System Design 1(2-3), 151–238 (1992)
9. Harrison, J.: Theorem Proving with the Real Numbers. Springer, Heidelberg (1998)
10. Hasan, O.: Formal Probabilistic Analysis using Theorem Proving. PhD Thesis, Concordia University, Montreal, QC, Canada (2008)
11. Hasan, O., Tahar, S.: Formalization of the Continuous Probability Distributions. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 3–18. Springer, Heidelberg (2007)
12. Hasan, O., Tahar, S.: Verification of Expectation Properties for Discrete Random Variables in HOL. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 119–134. Springer, Heidelberg (2007)
13. Hasan, O., Tahar, S.: Formal Verification of Tail Distribution Bounds in the HOL Theorem Prover. Mathematical Methods in the Applied Sciences (2008), http://www3.interscience.wiley.com/journal/120747455/abstract
14. Hurd, J.: Formal Verification of Probabilistic Algorithms. PhD Thesis, University of Cambridge, Cambridge, UK (2002)
15. ITRS (2008), http://www.itrs.net/links/2003itrs/home2003.htm
16. Kuo, S., Fuchs, W.K.: Efficient Spare Allocation for Reconfigurable Arrays. IEEE Design & Test of Computers 4(1), 24–31 (1987)
17. 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)
18. Low, C.P., Leong, H.W.: Probabilistic Analysis of Memory Reconfiguration in the Presence of Coupling Faults. In: Proceedings of the IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems (1992)
19. MacKay, D.J.C.: Introduction to Monte Carlo Methods. In: Learning in Graphical Models, NATO Science Series, pp. 175–204. Kluwer Academic Press, Dordrecht (1998)
20. Miczo, A.: Digital Logic Testing and Simulation. Wiley Interscience, Chichester (2003)
21. Nicolaidis, M., Achouri, N., Anghel, L.: A Diversified Memory Built-in Self-repair Approach for Nanotechnologies. In: Proceedings of the 22nd IEEE VLSI Test Symposium, pp. 313–318 (2004)
22. Rutten, J., Kwaiatkowska, M., Normal, G., Parker, D.: Mathematical Techniques for Analyzing Concurrent and Probabilisitc Systems. CRM Monograph Series, vol. 23. American Mathematical Society (2004)
23. Sehgal, A., Dubey, A., Marinissen, E.J., Wouters, C., Vranken, H., Chakrabarty, K.: Redundancy Modelling and Array Yield Analysis for Repairable Embedded Memories. IEE Proceedings of Computers and Digital Techniques 152(1), 97–106 (2005)
24. Sen, K., Viswanathan, M., Agha, G.: VESTA: A Statistical Model-Checker and Analyzer for Probabilistic Systems. In: Proc. IEEE International Conference on the Quantitative Evaluation of Systems, pp. 251–252 (2005)
25. Shi, W., Fuchs, W.K.: Probabilistic Analysis and Algorithms for Reconfiguration of Memory Arrays. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 11(9), 1153–1160 (1992)

Challenges in the Specification of Full Contracts*

Gordon J. Pace[1] and Gerardo Schneider[2]

[1] Department of Computer Science, University of Malta, Malta
[2] Department of Informatics, University of Oslo, Oslo, Norway
gordon.pace@um.edu.mt, gerardo@ifi.uio.no

Abstract. The complete specification of full contracts — contracts which include tolerated exceptions, and which enable reasoning about the contracts themselves, can be achieved using a combination of temporal and deontic concepts. In this paper we discuss the challenges in combining deontic and other relevant logics, in particular focusing on operators for choice, obligations over sequences, contrary-to-duty obligations, and how internal and external decisions may be incorporated in an action-based language for specifying contracts. We provide different viable interpretations and approaches for the development of such a sound logic and outline challenges for the future.

1 Introduction

The specification and analysis of *full contracts*, i.e. contracts between different entities regulating not only the normal interactive behaviours but also the exceptional ones, is becoming an imperative in many Computer Science applications. These include service-oriented architectures, e-commerce, component-based software development, and any other application where there is a need for trustful collaborative interactions. Such contracts should express not merely the sequence and causality of events, but also what are the *obligations*, *permissions* and *prohibitions* of the participating entities. These three notions being the object of study of the so-called deontic logic.

The specification of such contracts needs a formal language rich enough to capture deontic notions, temporal and dynamic aspects, real-time issues (e.g. deadlines), and the handling of actions (events) and exception mechanisms. Besides, contracts must be amenable to formal analysis techniques (e.g. model checking and runtime verification). Furthermore, the use of contracts is only meaningful if there is a mechanism to monitor their fulfillment. We believe that such successful language should be built on top of a basic deontic logic, in combination with modalities and operators from temporal, dynamic, action-based, and real-time logics.

Deontic logic is the logic concerned with moral and normative notions such as obligation, permission, prohibition, optionality, power, indifference, immunity and intention, among others. Though the scope of the logic (from the philosophical point of view) is huge and there is no way to formalise all those notions and study their relation in one single formalism, it is usually recognised that a deontic logic must contain at least the notions of obligation, permission, and prohibition, and preserve their intuitive

* Partially supported by the Nordunet3 project "COSoDIS".

properties. Even when restricting the logic to just these three notions, its formalisation is not easy, as witnessed by the extensive research conducted by the deontic community both from the philosophical and the logical point of view (see [19,31]).

Besides obligations, permissions and prohibitions, contracts, agreements, and normative systems contain clauses which by definition may be violated, represented by *contrary-to-duty obligations* (CTD) and *contrary-to-prohibitions* (CTP). CTDs are statements that represent the fact that obligations might not be respected where CTPs are similar statements which deal with prohibitions that might be violated. Both constructions specify the obligation/prohibition to be fulfilled and which is the *reparation/penalty* to be applied in case of violation.

We believe that when restricted to specific domains, deontic logic is a practical powerful specification tool, if combined with the above-mentioned logics. However, it is well known that deontic logic is not useful unless we ensure the absence of paradoxes and practical oddities [19,25]. Many of these paradoxes are due to the problematic combination of classical propositional logic operators with sequences, choices, repetitive behaviour, and CTDs and CTPs.

In this paper, we outline some of the choices to be made, and the challenges to be faced when combining deontic notions with other useful temporal concepts in the definition of a formal language for full contracts. Frequently, contracts are perceived simply as properties to be satisfied by a system. However, the analysis of the contracts as first class entities along the lines we are presenting it in this paper, can be fruitful in a variety of contexts especially in situations where services may require to be composed. Each service may come with its own contract, and the analysis of the composition of the constituent contracts may be necessary to study interaction of such services. One may also want to make queries about the contract, for instance to know what are the obligations of each participant and the penalties in case of not fulfillment of those primary obligations. Moreover, the decisions taken in the design of such a language are crucial since the ultimate goal is to have a framework facilitating contract analysis, as for instance: (i) Check that a particular system satisfies a contract; (ii) Reason about the contract itself directly; (iii) Reason about the relation between different contracts.

The paper is organised as follows. In the next section we argue for a ”practical” deontic logic, base for a formal contract language, highlighting general aspects. In section 3 we present our language of discourse for specifying full contracts. In the following sections we discuss the problems arising with the formalisation of obligation, CTDs, sequences, regular expressions and internal vs. external operators, in combination with other concepts from temporal, dynamic and real-time logics.

2 What Is Needed for a “Practical” Deontic Logic

As presented in the previous section, we claim that a good language for the specification of full contracts needs a practical deontic logic. The use of deontic logic goes beyond the standard formalisation of legal contracts, abstracting from other “modalities” and subtleties[1] and concentrating on obligations, permissions and prohibitions

[1] One may want to make a distinction between “rights” and “permissions”, between having the power to do something and the permission to do so, or even between “must”, “ought to” and “should”, but for certain kind of analysis these distinctions are not needed.

and their relation. Our aim is to restrict deontic logic to avoid paradoxes, and extend it accordingly as to be useful in the following contexts: (1) Fault-tolerant systems; (2) Compensable transactions; (3) Regulatory systems (4) Service-oriented architectures; (5) Component-based development.

Both fault-tolerant systems and systems with long transactions have in common that the specified desirable (*mandatory*) behaviour (sequence of states) will not necessarily be respected due to failures. In the presence of such failures, for some specified reason, we want to be able to come back to a previous state where an alternative behaviour must be enforced. This is very much what a CTD (or CTP) means. In some other cases we rather want to describe this deviation from expected mandatory behaviours as exceptions. Regulatory systems are normative systems containing regulations, laws and policies, rich on clauses specifying not only primary obligations but also exceptions. Such documents abound on cross-references to other clauses intra- and inter-document. In the context of SOA a deontic based approach may serve the purpose to give semantics to Service Level Agreements, or SLAs (which currently lack formal semantics) or to be used just as a specification language to write contracts between services. In component-based software development, contracts may be attached to components in order to guarantee, among other things, their compatibility both at development and deployment time.

Due to lack of space we will only indicate few papers where examples can be found. See for instance [26] for an example on the formalisation of a legal contract, and [10,7] for a justification on the use of deontic logic, and examples, on fault-tolerant systems. See [14,16] for compensable transactions, [11] for regulatory conformance checking, and [24] for state of the art and challenges in SOA. See [23] for a discussion of the use of contracts in the setting of component-based systems.

In order to avoid an inconvenient generalisation on the use of the term "deontic logic" we will restrict ourselves in what follows to a variant of deontic logic, which we believe is expressive enough to handle the representative number of applications mentioned above. We will call *OPP-logic* a logic or formal language containing the deontic modalities of Obligation, Permission and Prohibition (OPP), defined over complex actions, obtained from basic actions by (a restricted) combination of the following operators; choice $+$, sequential composition $\cdot$, concurrency $\&$, Kleene star $*$, and action negation $\bar{\cdot}$. Moreover, in OPP-logic it should be possible to specify: (i) Nested CTDs and CTPs; (ii) Temporal (causal) aspects; (iii) Nested Exceptions; (iv) Real-time aspects; (v) References to other expressions or clauses; (vi) Invariants; (vii) Fairness constraints; (viii) Contract introspection/reflection. Due to lack of space we will mainly concentrate on the first three aspects above, though we will also comment on the other items without entering into details.

3 The Language of Discourse

In this section, we outline the syntax of a language for the purpose of expressing full contracts. Please note that the language presented is not intended to be a complete formal language, but rather a language rich enough for us to illustrate issues in the use of a logic in expressing contracts over systems, in the spirit of the OPP-logic discussed above. The semantic issues will be discussed informally in later sections of the paper.

Actions: An important decision is that of whether the deontic operators act upon the state or the actions influencing the state. The two views, already familiar to computer scientists in the domain of specification languages[2], can both be defended, depending on the domain of application. We will base our approach on an action-based logic. As in the domain of process calculi, we go beyond simple actions to include parameterised actions (for example, *pay* may be a parameterised action whose parameter specifies the amount paid), and collections, or multisets of actions (to enable concurrent actions, including multiple instances of the same action). We also will use the notation $\overline{act}$ for any action different from act.

$$\text{Action} ::= \varepsilon \mid \text{Any} \mid \text{SimpAction} \mid \text{SimpAction(Param)} \mid \text{Action} \;\&\; \text{Action} \mid \overline{\text{Action}}$$

We will use lower-case Latin letters, $a, b, c, \ldots$ to denote basic actions.

Expressions over actions: Since we are interested in behaviour over time, to reason about causality, we require the enriching of actions over a temporal logic with operators such as sequentiality, choice and repetition. For the purposes of this paper, we will use extended regular expressions over actions, with + for choice, * for repetition, . for sequencing, & for concurrency and ¬ for negation.

$$\begin{aligned}\text{CompAction} ::= \;& \text{Action} \mid \neg\,\text{CompAction} \mid \text{CompAction}^{*} \mid \text{CompAction} \;+\; \text{CompAction} \\ \mid \;& \text{CompAction} \;\&\; \text{CompAction} \mid \text{CompAction} \,.\, \text{CompAction}\end{aligned}$$

We will use lower-case Greek letters, $\alpha, \beta, \ldots$ to denote compound actions.

Deontic operators: We will be using three basic deontic operators: permission, prohibition and obligation, which can be applied on compound actions. The basic contracts $\mathbb{Y}$ and $\mathbb{N}$, respectively corresponding to the trivially satisfied and unsatisfiable contracts, will also be included for completeness.

$$\text{SimpContract} ::= \mathbb{Y} \mid \mathbb{N} \mid \mathbb{P}(\text{CompAction}) \mid \mathbb{F}(\text{CompAction}) \mid \mathbb{O}(\text{CompAction})$$

Default contracts: Rather often, contracts are composed of a cascade of contracts: you are obliged to do something, but if you do not, you are then obliged to do something else. In general, one may define a defaulting operator over contracts which, given two contracts, behaves like the first, but enacts the second if the first is broken. However, we will limit ourselves to two simple forms of this operator in this paper: to Contrary-To-Duty (CTD) contracts, and Contrary-To-Prohibition (CTP) contracts. A CTD is made up of a compound action, and another contract — the performance of the action is obliged, but if not performed, the contract is enacted. Similary CTPs enact a prohibition by default. In other cases we may rather consider exceptions, where a given contract is not enforced if another one is fulfilled.

$$\begin{aligned}\text{CompContract} ::= \;& \text{SimpContract} \mid \text{CTD(CompAction, CompContract)} \mid \\ & \text{CTP(CompAction, CompContract)} \mid \\ & \text{CompContract} \; \mathit{unless} \; \text{CompContract}\end{aligned}$$

Expressions over contracts: We also require temporal operators over contracts (for instance, to be able to express that an obligation is enacted as soon as another obligation is satisfied). We will similary use regular expressions for this purpose.

[2] Specification languages such as Z, enable a state-based view of the system, as opposed to process calculi such as CSP and CCS, where the emphasis is on the actions.

We add another operator to test for the presence of an action, with $a?.C$ being equivalent to contract C (as of the next time unit) if a is present now, but void (for which nothing needs to be done to satisfy it) otherwise.

$$\begin{aligned}\text{Contract} ::= {} & \text{CompContract} \mid \neg\text{Contract} \mid \text{Contract}^* \mid \text{Contract} \ + \ \text{Contract} \mid \\ & \text{Contract} \ \& \ \text{Contract} \mid \text{Contract} \,.\, \text{Contract} \mid \text{CompAction}? \,.\, \text{Contract}\end{aligned}$$

The use of contract negation may seem an odd choice — the interpretation of negation is that the particular contract is not enacted. For example, one can express permission to perform an action, as the negation of the obligation to perform something other than the action. We will not provide a formal semantics of the language since our intention is to explore the design and problems of such formal language and not its particular formalisation.

4 Combining Temporal and Deontic Notions

4.1 Sequences

We need sequences of actions if we want to distinguish between situations such as "You are obliged to fill in the form, and then sign it" from "You are obliged to fill in the form, after which you are obliged to sign it".

Sequences over Contracts, and Contracts over Sequences. The choice of including sequences both inside and outside contracts is arguably necessary for a number of reasons. The semantic difference between a statement such as $\mathbb{F}(a.b)$ and $\mathbb{F}(a).\mathbb{F}(b)$ is rather straightforward — while the former disallows the *sequence* of a followed by b, the latter forbids a, after which it forbids b. While the former allows an action a, followed by c, the latter is broken upon the arrival of action a. The distinction is similarly clear with permissions. However, the distinction when it comes to obligations of sequences is finer [32]. In DDL (Dynamic Deontic Logic) sequences of obligations (SoO) and obligations on sequences (OoS) are equivalent [21], though they should not be considered equivalent [32]. This is indeed clear if we consider what is the consequence of fulfillment and violation of such contracts. In SoO one can associate different reparations to each element on the sequence, while this is not the case in OoS (see [32] for examples). The need for distinguishing between sequences of contracts and contracts over sequences is however justified on a number of criteria.

Elegance of expression: It can be argued that contracts over sequences can be encoded as sequences over contracts. For instance, $\mathbb{F}(a.b)$ can be (up to a certain extent) be encoded as $a?.\mathbb{F}(b)$, similarly $\mathbb{O}(a.b)$ could be written as $\mathbb{O}(a) \wedge a?.\mathbb{O}(b)$. If a contract's semantics cares only about the instances of failure (breaking the contract), the contracts could be argued to be similar. However, the rewritten contracts (as sequences of contracts) lose the direct intuitive meaning originally meant and expressed as contract over sequences. Furthermore, without introducing a general contract default operator, expressing CTDs and CTPs over contracts on sequences introduces further complications — for example, CTD$(a.b, C)$ could be rewritten into something of the form $a?.\text{CTD}(b, C) \wedge \bar{a}?.C$.

Contract violation: If a prohibition clause over a sequence $a.b$ is expressed using a conditional term $a?.\mathbb{F}(b)$, the first action a is not seen as an explicit part of the prohibition.

The interpretation of such a term indicates that breaking the contract results from performing action b which is forbidden after an action a. This is in direct contrast with the intended interpretation which forbids the sequence $a.b$.

Reasoning about contracts: Furthermore, the failure semantics of a contract is but one interpretation. Other views of the contract may include analysis directly related to the deontic operators in a contract (such as which contracts are active at a particular point in time, the total number of obligations in a particular contract, etc). Furthermore, if one extends the deontic language to refer to use the presence or absence of obligations and prohibitions as conditions within the language, action sequences become indispensable.

Causality. Another interesting issue is that of causality. Consider the semantics of the default contract operators such as CTD(α, C). The informal interpretation of the operator is that an obligation to perform α is enacted, but if it is not, contract C has to be satisfied. However, this leads to two differing views of the operator: (i) contract C must hold as soon as (or one time unit after) the initial obligation is broken; or (ii) the choice between performing the obligations or the alternative contract C as soon as the CTD is enacted. In the literature, the first interpretation is typically given. The second interpretation leads to interesting causality situations. Consider the contract CTD$(\text{Any}.a, \mathbb{O}(b))$ — if one views $\mathbb{O}(b)$ as an exception/option, it has to be done before breaking the original obligation. An initial action b may satisfy the CTD or not — there is no way we can know this until we get the second set of events.

Breaking an obligation. CTDs and contracts with exceptions still prove to be challenging with sequences of actions. Consider the following situation:

> *The law of a country says that: 'You are obliged to hand in Form A on Monday and Form B on Tuesday, unless officials stop you from doing so.'*
> *On Monday, John spent a day on the beach, thus not handing in Form A. On Tuesday at 00:00 he was arrested, and broght to justice on Wednesday.*
> *The police argue: 'To satisfy his obligation the defendant had to hand in Form A on Monday, which he did not. Hence he should be found guilty.'*
> *But John's lawyer argues back: 'But to satisfy the obligation the defendant had to hand in Form B on Tuesday, which he was stopped from doing by officials. He is hence innocent.'*

Who is right? Formalising the primary obligation in the law, we get $\mathbb{O}(a.b)$, where a represents handling Form A on Monday and b handling Form B on Tuesday. When is the obligation to be considered violated — upon the lack of action a, or at the end of two consecutive actions? It will depend on whether we model the above with CTDs or "unless", and what is the formal semantics. To avoid this and similar paradoxes, we propose the use of an earliest failure semantics, in which, a contract fails as soon as it can no longer be satisfiable.

CTDs and Sequences of Actions. The introduction of time into the deontic soup raises the question of what are the most natural semantics to CTDs. For example, let us consider CTD$(a, \mathbb{O}(b))$. Does this correspond to an obligation to do a, which *if violated*, will then set up an obligation to perform a b, or can the b be performed immediately to satisfy the contract. In other words, does the sequence of actions $(\bar{a}\&\bar{b}).b$ satisfy the contract? What about $\bar{a}\&b.\bar{a}$?

The enactment of the compensation contract (and hence the possibility of satisfying it) only after the first obligation is violated distinguishes CTDs from mere choice, even if external choice over contracts may be desirable in certain contexts. This will be further discussed in the coming sections.

Introducing sequences of actions into a CTD is directly related to the paradox given above. If the moment of violation of the obligation is used as the triggering of the new (compensation) contract, earliest failure semantics may be necessary.

4.2 Choice

The choice operator has an intuitively different semantics when given inside, or outside a deontic operator. The contract obliging you to hand in Form A or Form B, is distinct from the disjunctive contract which says that either you are obliged to hand in Form A, or you are obliged to hand in Form B. The interpretation of the latter contract has been much discussed in the literature, and various paradoxes are known, which challenge various naïve semantics of the operator. In our opinion, it is not advisable to have the classical disjunction in contracts, or at least to restrict its use, since De Morgan rules may introduce paradoxes.

Choice of Obligations, and Obligations of Choices. Let us start by considering the contract with an obligation over a choice: $\mathbb{O}(a+b)$. As in standard process calculi, the choice operator can have (at least) two radically different views: angelic vs demonic, or sometimes internal vs external choice. Does performing action a always satisfy the contract, or is the choice made internally (demonically) and you may be obliged to perform action b? Similarly, the choice over contracts has a similar issue — $\mathbb{O}(a) + \mathbb{O}(b)$. The distinction between the two as used in natural language contracts can be made clearer by considering financial contracts which John signs with Peter:

Contract 1: 'On the 1st of May, John will either (i) be obliged to sell 100 shares at $1 each; or (ii) be obliged to sell 50 shares at the market price.'

Contract 2: 'On the 1st of May, John will be obliged either (i) to sell 100 shares at $1 each; or (ii) to sell 50 shares at the market price.'

The (possibly debatable) interpretation of the contracts is that, while in contract 1 the choice of which obligation to enact lies with Peter, in the latter one obligation is enacted, and it is up to John to decide how to discharge it. Peter should prefer the first contract, whereas John should prefer the second.

This interpretation corresponds to having choice outside the deontic operators to be resolved by the contract controller (demonic, or internal choice), while choice inside deontic operators to be resolved at the entity upon whom the contract acts (angelic, or external choice).[3] The only way to, *a priori*, guarantee satisfying the contract $\mathbb{O}(a)+\mathbb{O}(b)$ is through performing both actions a and b, which may be impossible if the semantics of $+$ is given as an exclusive or.

[3] Note that here internal and external refers to the contract, and not to the performer of the action.

This raises the question, once again, whether the logic, or implementation should have access to currently enacted obligations. In such cases, one would be able to write something on the lines of $O_a?.a + O_b?.b$ ($O_a?$ says whether an obligation to do a is active right now, continuing without any delay in time) to guarantee that the contract $\mathbb{O}(a) + \mathbb{O}(b)$ is always satisfied. On the other hand, having access to this information within the logic may lead to contructive anomalies with contracts such as $\neg O_a?.\mathbb{O}(a)$. Constructiveness of such logics has been studied in other contexts such as synchronous programming [28] and cycles in circuits [9].

Let us consider now the contract: You are forbidden from doing a or b, written as $\mathbb{F}(a + b)$. In $\mathbb{O}(a + b)$ we seem to take it to mean 'you must do something which creates a trace satisfying the regexp $a + b$'. In $\mathbb{F}(a + b)$, we are saying 'you must not do something which creates a trace satisfying the regexp $a + b$'. What is the meaning of $+$ in $\mathbb{F}(a + b)$? It seems like the choice inside a forbidden operator becomes an internal choice, not an external one. The implicit negation outside the $\mathbb{F}$ switches the $+$ from external to internal. Is this a suitable interpretation? It depends. If we define $\mathbb{F}(a + b)$ to be $(\mathbb{F}(a) \wedge \mathbb{P}(b)) + (\mathbb{F}(b) \wedge \mathbb{P}(a))$ it seems that the above interpretation is correct. On the other hand, we have a different interpretation if we consider that $\mathbb{F}(a + b)$ to be defined as $\neg\mathbb{P}(a + b)$, where $\mathbb{P}(a + b)$ means the same as $\mathbb{P}(a) \wedge \mathbb{P}(b)$, in which case we get that both a and b are forbidden.

Clearly, it is debatable whether external and internal choice should be separate operators or a single operator with different interpretations in different modalities.

The Moment of Choice and the Moment of Contract Satisfaction. Differentiating between internal and external choice makes it possible to express different contracts in a natural way. However, another parameter which has to be decided for a temporal deontic logic is *when* the choice is made. Consider the non-deterministic contract $\text{Any}?.\mathbb{O}(a) + \text{Any}?.\mathbb{O}(b)$. In this example, Any? acts like the silent action τ in process calculi such as CCS [22], and raises the question of whether this contract is in any way different from $\text{Any}?.(\mathbb{O}(a) + \mathbb{O}(b))$. The choice between contracts can be made immediately, or later on in the future, as soon as a contract is to be enacted. Therefore, to write a contract which leads to different obligations depending on the presence (or otherwise) of a particular action, one would express it as: $(a?.\mathbb{O}(b)) + (\bar{a}?.\mathbb{O}(c))$.

On the other hand, due to the user-centric view of angelic choice, the satisfaction of a contract with choice may yield different interpretations. Consider a contract such as $\mathbb{O}(a + a.c).\mathbb{O}(d)$. After receiving an action a, we are unsure whether the first contract has been satisfied, since it depends on whether the user will proceed with $c.d$, or simply with d. Similarly, given the contract $\mathbb{O}(a{+}a.c).\mathbb{O}(c)$, it is non-deterministic whether the action sequence $a.c.c$ satisfied the contract after the first two, or three symbols. Besides the above technicalities concerning non-determinism, we argue for forcing deterministic contracts, as desirable both in the legal arena as in our mentioned applications.

Choice and CTDs. If choice between contracts leads to internal, nondeterministic choice, a contract such as $\text{CTD}(a, b) + \text{CTD}(c, d)$, may be broken if one performs an action a but no c (and no d to compensate). Without having access to how the choice over contracts is resolved, leads to having to satisfy both contracts. With CTDs, such a composition of contracts may lead to unsatisfiable contracts in roundabout, counter-intuitive ways. Furthermore, the issue of when a contract over sequences is violated rises again, since the CTD is triggered upon violation.

Note that the choice operator may be seen as a kind of exclusive disjunction. However, this similarity is only apparent as clear in the case of writing the following contract: $\text{CTD}(a, \mathbb{O}(b)) + \text{CTD}(b, \mathbb{O}(a))$. With an exclusive or point of view, the above will lead to a violation independently of what is done, as performing a may be seen as satisfying the primary obligation in the first disjunct, but also the reparation in the second; similarly with b. The above example is problematic even under the interpretation of $+$ as choice, if non-determinism is allowed.

4.3 Repetition

The use of repetition in contracts, corresponding to a combination of choice, sequences and fix-points, poses a variety of challenges related to the ones already discussed. As in the case of choice and sequences, a contract stating that 'John is repeatedly obliged to pay after which he is permitted to use the service', is different from 'John is obliged to pay repeatedly after which he will be permitted to use the service' — $\mathbb{O}(p)^*.\mathbb{P}(s)$ as opposed to $\mathbb{O}(p^*).\mathbb{P}(s)$.

Notation. Different approaches to logic use different operators for repetition. Languages such as $\mathcal{CL}$ [26] use the LTL- and CTL-style until operator. In this paper, we use the regular expression-style star operator to indicate repetitions. This gives a uniform view of temporal operators inside, and outside the deontic operators, and enables repetition of contracts which take more than one time unit to terminate. For instance, the contract $\mathbb{O}(5c.10c + 50c)^*.\mathbb{P}(choc)$ will repeatedly obliges inserting 5 and 10 cent coins in sequence or a 50 cent coin, until at the end withdrawing a chocolate is permitted. Similarly, one can give a contract $\mathbb{O}((5c.10c + 50c)^*).\mathbb{P}(choc)$, which has a single obligation for the user to insert any number of coins, after which she is permitted to withdraw a chocolate. Should the two contracts be distinguishable?

Expressing contracts with action sequences within the deontic operators using an until operator can prove to be challenging. It is certainly desirable to have the inner temporal logic match (at least in style) with the outer one. Using an interval logic (regular expressions) inside the deontic operators and a point logic outside the operators can result in spin the presence of CTDs (and CTPs).

Contracts of Repetitions and Repetitions of Contracts. Informally, equating the star operator with an unbounded sequence of choices indicates that the interpretation of the choice operator inside and outside the deontic operators should be respected with repetition. In other words, a contract such as $\mathbb{O}(a^*)$ (intuitively equivalent to $\mathbb{O}(\varepsilon + a + a.a + \ldots)$) means that a number of actions a are to be performed — the choice regarding the number of repetitions is external, that is, decided by the entity bound by the contract. On the other hand, with $\mathbb{O}(a)^*$ (intuitively equivalent to $\mathbb{Y} + \mathbb{O}(a) + \mathbb{O}(a).\mathbb{O}(a) + \ldots$), the choice regarding the number of repetitions is internal, and thus imposed.

Unbounded Repetition. Let us consider the contract saying that 'If John uses the service, then he is bound to eventually pay.' One would write this as $s?.\mathbb{O}(\text{Any}^*.p)$. Note that no bound is placed on how long John takes to pay his dues. Giving a formal semantics of the logic over infinite sequences enables one to say whether or not John has satisfied the contract. On the other hand, when one looks at finite sequences, one requires the use of a three-valued logic to differentiate between the contract being violated, satisfied, and the third situation when it may still be satisfied in the future. In certain contexts, such as runtime verification, this approach will be required. In practice, however, it seems more natural to have only bounded iteration, to be able to enforce the payment in a life-time period. However, one may question whether unbounded repetition *inside* deontic operators is meaningful. John would certainly have no qualms about signing the above-mentioned contract, since he is not due to perform any action on his part to satisfy the contract. On the other hand, unbounded repetition outside deontic operators is still meaningful, given that the choice over repetition is an internal one.

4.4 Other Issues

Real-Time Aspects. Most useful contracts include some timing aspects: deadlines, timeouts, durations, etc. Dealing with real-time introduces further challenges when combined with deontic notions: Should we associate time with the modalities, clauses, actions, or with all of them? Is an interval-based logic necessary to reason about the beginning and end of an action?

Since we want not only to specify but also to be able to analyse contracts written in our logic, we aim at a decidable extension. Though many of the above decisions may be application-driven, we propose to consider the use of *clocks* with freezing quantifiers and resets [1]. A modular conservative approach would be to extend an untimed OPP-logic with clocks. In this way a suitable combination of Kripke-like structures with timed automata could be envisaged as a semantical framework for such a logic.

Reference to other expressions. A *nominal* logic or simply annotations on clauses and contracts may be needed to be able to refer to other clauses (in the same contract) or contracts. In principle the analysis of cross-references could be analysed with standard existing techniques on graph (e.g. reachability analysis).

Introspection / reflection. One interesting extension is that of introducing contract introspection — the capability of having conditions which depend on which obligations, permissions and prohibitions are active, to express contracts such as 'Whenever you are obliged to pay, you are also obliged to produce identification'. Furthermore, a contract may contain references to itself, i.e. be *reflexive*, for instance a clause may determine that under certain circumstances a party may have the *power* to change other clauses, or even to cancel the contract. Note that, not only does this complicate compositional analysis of contracts, but may also introduce causality issues. Related to contract introspection, we could also envisage to have a formal theory to relate *policies* and contracts (i.e. "vertical" contracts regulating "horizontal" contracts).

Fairness and invariants. Other issue is how easy is to represent *fairness* and *invariants*. In what concerns invariants, we may be able to represent the "box" operator of LTL in a similar way as it is defined in Duration Calculus, i.e. by negating a particular contract sequence. Concerning fairness, we want to specify properties like "any infinitely often enabled process should be infinitely often taken" (or other variations of fairness constraints usually defined in temporal logics). As in temporal logics, we may decide to take a syntactic approach (e.g. as in LTL) in which case the logic should allow writing something like $\Box\Diamond C$, or a semantic approach (e.g. *à la* CTL). Again, in practice it seems reasonable to enforce only bounded fairness.

Concurrency. In many applications it seems natural to consider true concurrency, mainly under the deontic operators. For instance, one may need to say that 'you are obliged to sit-down and remain silent', $\mathbb{O}(s\&r)$, where an eventual violation of this obligation includes not doing any (nor both) actions. The combination of such concurrent operator introduces many challenges and its combination with the other regular expression operators is not easy. One problem is that depending on the interpretation of conjunction on obligations, it may be difficult to assert whether $\mathbb{O}(s) \wedge \mathbb{O}(r)$ entails $\mathbb{O}(s\&r)$, or whether they are equivalent. In the latter this introduces a big challenge concerning the semantic interpretation of $\&$ in a deontic logic based on actions in the style of dynamic logic, since conjunction is usually interpreted as a branching in certain cases. See [27] for further discussion on the topic.

Conditional Contracts. The use of conditions outside the deontic operators enables the formulation of contracts which are dependant on runtime behaviour. The question of whether such conditions should also be allowed inside the deontic operators leads to interesting possibilities. Consider the contract 'Unless the service is disabled, John is obliged to pay in the next time unit' as opposed to the contract 'John is obliged to pay in the next time unit, unless the service is disabled now.' The former obligation is never enacted if the service is disabled, whereas the latter is enacted, but becomes trivially satisfied when disabling the service. The former corresponds to $\bar{d}?.\mathbb{O}(p)$, while the latter to $\mathbb{O}(\bar{d}.p + d)$ (or even Any.$\mathbb{O}(p)$ $unless$ $d?$). Different approaches can be used to express such conditional contracts.

Encoding as normal actions: As seen in the example, one can encode conditional contracts using regular expressions. The main drawback of this approach is that which actions are conditions, and which are actually obliged by the contract becomes blurred. Although for most uses of a contract such a view is sufficient, when analysing a contract (for instance, to identify what actions one may be obliged to take) one may require further information.

Subjects, objects and actors: One approach is the identification of the actors of a contract — who the subject of an action is, and who the object is. Automatically, actions which are not performed by the party being obliged to do something, are conditions. However, this approach fails when the condition includes actions under the control of the party. In the above example, John may be the actor who chooses whether or not to disable the service. Introducing the subject and object of an action gives insight into a contract, but not sufficient information to analyse conditions.

Assertions: Actions (or regular expressions over actions) may be explicitely tagged as conditions, or assertions to indicate that they are not part of the obligation or prohibition: $\mathbb{O}(d? + \bar{d}?.p)$. The semantics to such a statement, is that if d is not present, then p must follow, otherwise the obligation becomes trivially violated. Although one may also give a semantics in which the default is the trivially satisfied contract, this leads to confusing situations — consider the contract 'John is obliged to pay until he disables the service'. This can be written as $\mathbb{O}((\bar{d}?.p)^*d?)$, which intuitively corresponds to $\mathbb{O}(d? + \bar{d}?.p.d? + \bar{d}?.p.\bar{d}?.p.d? + \ldots)$. Recall that we suggested that a logical choice for the semantics of choice inside deontic operators is an external (user-driven) choice. But by not disabling the service and choosing the first choice in the unrolled version of the contract, the contract is trivially satisfied. Rewriting the contract to work with such semantics would require an implicit choice operator within deontic operators, or use unnecessarily intricate formulations such as: $\mathbb{O}((d?.0 + \bar{d}.p)^*(d? + \bar{d}.0))$ (where 0 corresponds to an action that can never be performed).

Another choice in the design of conditions in a logic for contracts is that of whether they should take time to evaluate. If the original contract discussed in this section were 'John is obliged to pay, unless the service is disabled,' one would have to reformulate the formal contract using conjunction $\mathbb{O}((\bar{d}?\&p) + d?)$. An alternative is to have conditions (both inside and outside deontic operators) with no time to evaluate, and proceed in the same time unit. Having such an approach makes expressing certain properties more straightforward, but at the cost of the need for constructiveness checking to avoid expressing counter-intuitive, or meaningless properties such as $\bar{d}?.d$ (if d is not present in the current time unit, then ensure it is).

5 Final Discussion

We have argued here that a formal language for full contracts needs a 'practical' deontic logic in combination with features from many other logics, and we have discussed some problems in obtaining such a logic. Giving a sound formal semantics to a real-time temporal deontic logic is not an easy task. Different approaches may be necessary for different application areas. To be able to discuss CTDs and CTPs (or default contracts, in general), a failure-oriented semantics is very desirable. Equivalence of contracts is another major challenge, and a bisimulation based approach can prove to be fruitful — the main question is to define reasonable bisimulation relations to encompass the different views of what equivalence over contracts means. Analysis of contracts is another important area worthy of investigation. We have recently developed a model-checking technique for the discovery of contradictory contracts in $\mathcal{CL}$ [13]. However, other analysis techniques are required — constructiveness analysis ensures no causality cycles in a contract, timing analysis may be used to study the lifetime of a contract, etc.

In identifying a number of challenges in combining temporal and deontic operators, we have mentioned various extensions which introduce more expressivity, but also further complexity into the logic. These include introspection, real-time issues, cross-references, a suitable treatment of true concurrency, fairness and conditional contracts

Also, since we have considered an *ought-to-do* approach (i.e. the deontic modalities are applied on actions and not state-of-affairs), one may also have to consider many aspects from dynamic logic, in the spirit of Meyer's work [21] (see also [4,26]).

Finally, we have used a simple temporal logic (regular expressions) onto which to graft our deontic logic. One may choose to start from another logic such as LTL, CTL or μ-calculus. The general question of how one can uniformly extend a (discrete) temporal logic with deontic operators is an interesting one, giving a unified view of a *deontic transformer* (in the style of monad transformers [17] in functional programming).

Besides specific technical differences with problems identified in previous work on deontic logic (see related work below), our work is set apart by the fact that some challenges in the specification of full contracts go beyond the "traditional" research by the deontic community. Thus, the definition of a formal language (and reasoning system) for full contracts in Computer Science applications, involves additional challenges, as for instance: (1) Combination with other logics, in particular real-time logics; (2) The need of a *trace semantics* for the contract language, since contracts must be monitored at runtime; (3) An algorithm to automatically obtain such a runtime monitor; (4) Eventual use of an enforcement mechanism, at runtime; (5) Development of a *contract-as-types* theory in order to manipulate contracts at the programming language level.

Related Work. Puzzles and paradoxes have accompanied deontic logic since its very beginning [30], starting with its formalisation in the so-called standard deontic logic (SDL). See von Wright's account [31] and McNamara's article [19]

Problems in formalising CTDs in the original papers on deontic logic were first discussed by Chisholm in [8]. Åqvist [2] gave a solution to the paradoxes related to CTDs (and also to the Good Samaritan and the paradox of the epistemic obligation) by proposing different semantic relations for primary and reparational obligations. The problem with the proposed solution is that one might need an unbounded (and eventually infinite) number of such relations in case of an unbounded number of nested CTDs. Prakken and Sergot [25] further discussed the difficulties to get a good representation of CTDs, based on an ought-to-be approach. In particular they showed how non-monotonic methods are not suitable for such cases, since such logics (e.g. *defeasible* logic) treat CTDs as exceptions, in which case the primary obligation is not really violated, since it is never applied. The difference between a CTD and an exception may be seen in the following example: Let A be 'you are obliged to pay unless somebody else pays for you', and B be 'you are obliged to pay, and if you do not pay somebody else has to pay for you'. A expresses an exception where 'somebody else pays for you' can be seen as a rule *defeating* the first obligation, which is never enacted in case the exception happens. However, in B, whenever you do not pay and somebody else does it for you, there has been a violation of the primary obligation, and the fact that somebody pays for you is seen as a reparation to the violated obligation. The difference is not naïve, and contracts A and B should be seen as different.

The problem of sequence of obligations versus obligation on sequences, and their combinations with CTDs has been discussed by Wyner in [32]. Wyner argues that none of previous work (including [6,21]) have given a good unproblematic representation of CTDs, in particular its relation with sequences and obligations. Khosla and Maibaum [15] mentioned these differences but did not propose a suitable solution. Wyner further studies the distinction, making a difference between 'distributed obligations´ and 'obligations on an interruptible sequence', corresponding to our proposal of external and internal sequences on obligations.

A discussion on the difference between internal and external choice in deontic logic has been addressed for instance in [20] and more recently in [29].

In [18] Lomuscio and Sergot describe a way to distinguish between normal and exceptional cases in their deontic interpreted systems, from the semantical point of view, by separating "allowed" from "disallowed" states.

A different approach based on dynamic logic by Meyer [21], where a special *marker* V for violation is introduced in order to mark that an obligation has been violated (or a forbidden action has been performed) in the current world (where V holds). Though the approach makes a big step towards a solution of many of the deontic paradoxes, it does not properly address the problem of what follows after a violation. Furthermore, in Meyer's logic it is a theorem that sequences of obligations are equivalent to obligation on sequences, which is not the case as we have argued in section 4.1.

An example of the use of a restricted deontic logic for fault-tolerant systems is presented by Castro and Maibaum [7] (see also Coenen's work [10]). As far as we know, there is no application of deontic logic to systems with compensable transactions. Most of the research conducted in this domain uses process calculi (see for instance [16,5]).

A choice operator explicitly appears in those works using the ought-to-do approach, where the operator is among actions ($a + b$), for instance in the works by Broersen et al [4], and Prisacariu and Schneider [26]. The latter also allowing concurrent actions.

A logic with deontic flavor has been recently introduced by Dinesh et al. in [11,12]. The main objective of that work is to represent conditional obligations and permissions, to capture exceptions to norms, and most notably to be able to represent (and reason) about references among norms (clauses). As far as we know, it is not possible to represent CTDs and CTPs in that logic. However, the application domain of the paper is the regulatory system of blood donations, where CTDs and CTPs are not common.

The recent paper by Åqvist [3] proposes a formalisation of a logic for conditional obligations and permissions, combined with temporal modalities. No CTDs, CTPs, nor real-time is considered.

Finally, note that the concept of *contract* used in this paper is more general than behavioural interfaces, the design-by-contract programming style using pre- and post-conditions, and other more "standard" notion of contracts in Computer Science (see [23] for a discussion on that).

References

1. Alur, R., Henzinger, T.A.: Real-time logics: complexity and expressiveness. Information and Computation 104, 390–401 (1993)
2. Åqvist, L.: Good samaritans, contrary-to-duty imperatives, and epistemic obligations. Noûs 1(4), 361–379 (1967)
3. Åqvist, L.: Combinations of tense and deontic modality: On the approach to temporal logic with historical necessity and conditional obligation. J. Applied Logic 3(3-4), 421–460 (2005)
4. Broersen, J., Wieringa, R., Meyer, J.-J.C.: A fixed-point characterization of a deontic logic of regular action. Fundam. Inf. 48(2-3), 107–128 (2001)
5. Bruni, R., Melgratti, H.C., Montanari, U.: Theoretical foundations for compensations in flow composition languages. In: POPL 2005, pp. 209–220. ACM Press, New York (2005)
6. Carmo, J., Jones, A.J.: Deontic logic and contrary-to-duties, vol. 8, pp. 265–343. Kluwer Academic Publishers, Dordrecht (2002)
7. Castro, P.F., Maibaum, T.S.E.: A complete and compact propositional deontic logic. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) ICTAC 2007. LNCS, vol. 4711, pp. 109–123. Springer, Heidelberg (2007)

8. Chisholm, R.M.: Contrary-to-duty imperatives and deontic logic. Analysis (XXIV), 33–36 (1963)
9. Claessen, K.: Safety Property Verification of Cyclic Synchronous Circuits. In: SLAP 2003. ENTCS, vol. 88. Elsevier, Amsterdam (2003)
10. Coenen, J.: Top-down development of layered fault tolerant systems and its problems- a denotic perspective. Ann. Math. Artif. Intell. 9(1-2), 133–150 (1993)
11. Dinesh, N., Joshi, A., Lee, I., Sokolsky, O.: A logic for regulatory conformance checking. In: Proceedings of the 14th Monterey Workshop (2007)
12. Dinesh, N., Joshi, A., Lee, I., Sokolsky, O.: Reasoning about conditions and exceptions to laws in regulatory conformance checking. In: van der Meyden, R., van der Torre, L. (eds.) DEON 2008. LNCS, vol. 5076, pp. 110–124. Springer, Heidelberg (2008)
13. Fenech, S., Pace, G.J., Schneider, G.: Conflict analysis of deontic contracts. In: WICT 2008 (November 2008) (to appear)
14. Hoare, C.A.R., Butler, M., Ferreira, C.: A trace semantics for long running processes. In: Abdallah, A.E., Jones, C.B., Sanders, J.W. (eds.) Communicating Sequential Processes. LNCS, vol. 3525, pp. 133–150. Springer, Heidelberg (2005)
15. Khosla, S., Maibaum, T.S.E.: The prescription and description of state based systems. In: Banieqbal, B., Pnueli, A., Barringer, H. (eds.) Temporal Logic in Specification. LNCS, vol. 398, pp. 243–294. Springer, Heidelberg (1989)
16. Li, J., Zhu, H., Pu, G., He, J.: A formal model for compensable transactions. In: ICECCS 2007, pp. 64–73. IEEE Computer Society Press, Los Alamitos (2007)
17. Liang, S., Hudak, P., Jones, M.: Monad transformers and modular interpreters. In: POPL 1995, pp. 333–343. ACM Press, New York (1995)
18. Lomuscio, A., Sergot, M.: Deontic interpreted systems. Studia Logica (75), 63–92 (2003)
19. McNamara, P.: Deontic logic. Handbook of the History of Logic, vol. 7, pp. 197–289. North-Holland Publishing, Amsterdam (2006)
20. Meyer, J.-J.: Free choice permissions and ross's paradox: Internal vs. external nondeterminism. In: Proc. 8th. Amsterdam Collloquium, University of Amsterdam, pp. 367–380 (1992)
21. Meyer, J.-J.C.: A different approach to deontic logic: Deontic logic viewed as a variant of dynamic logic. Notre Dame Journal of Formal Logic 29, 109–136 (1988)
22. Milner, R.: A Calculus of Communicating Systems. Springer, New York (1982)
23. Owe, O., Schneider, G., Steffen, M.: Components, objects, and contracts. In: SAVCBS 2007, Dubrovnik, Croatia, September 2007. ACM Digital Library, pp. 91–94 (2007)
24. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-oriented computing: State of the art and research challenges. Computer 40(11), 38–45 (2007)
25. Prakken, H., Sergot, M.: Contrary-to-duty obligations. Studia Logica 57(1), 91–115 (1996)
26. Prisacariu, C., Schneider, G.: A Formal Language for Electronic Contracts. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS, vol. 4468, pp. 174–189. Springer, Heidelberg (2007)
27. Prisacariu, C., Schneider, G.: Towards a formal definition of electronic contracts. Technical Report 348, Dept. of Informatics, Univ. of Oslo. (January 2007)
28. Shiple, G.B.T., Touati, H.: Constructive analysis of cyclic circuits. In: European Design and Test Conference (1996)
29. van der Hoek, W., van Linder, B., Meyer, J.-J.C.: On agents that have the ability to choose. Studia Logica 66(1), 79–119 (2000)
30. Wright, G.H.V.: Deontic logic. Mind 60, 1–15 (1951)
31. Wright, G.H.V.: Deontic logic: A personal view. Ratio Juris 12(1), 26–38 (1999)
32. Wyner, A.Z.: Sequences, obligations, and the contrary-to-duty paradox. In: Goble, L., Meyer, J.-J.C. (eds.) DEON 2006. LNCS, vol. 4048, pp. 255–271. Springer, Heidelberg (2006)

Partial Order Reduction for State/Event LTL

Nikola Beneš[*], Lubos Brim[*], Ivana Černá[**], Jiri Sochor[**], Pavlina Vařeková[**], and Barbora Zimmerova[**]

Faculty of Informatics, Masaryk University, Brno, Czech Republic

Abstract. Software systems assembled from a large number of autonomous components become an interesting target for formal verification due to the issue of correct interplay in component interaction. State/event LTL [1,2] incorporates both states and events to express important properties of component-based software systems.

The main contribution of the paper is a partial order reduction technique for verification of state/event LTL properties. The core of the partial order reduction is a novel notion of stuttering equivalence which we call state/event stuttering equivalence. The positive attribute of the equivalence is that it can be resolved with existing methods for partial order reduction. State/event LTL properties are, in general, not preserved under state/event stuttering equivalence. To this end we define a new logic, called weak state/event LTL, which is invariant under the new equivalence.

1 Introduction

Increasing complexity in software development stimulates application of new techniques that help to deliver systems in shorter time and with lower costs. One of such techniques is the component-based development, that builds software systems out of prefabricated autonomous components, often developed with no knowledge of their deployment context. Under such conditions, interaction among components in the system becomes a crucial issue in the system correctness.

Verification of component-based systems. Similarly to communicating processes, interaction of components can be formalized in terms of labelled transition systems, representing communicational behaviour of the components, and correctness of the systems in a temporal logic. In practice, real systems are composed of a large number of components which are often independent on each other and run concurrently. In such cases, automated verification becomes challenging due to their size and complexity. This motivates the search of component-specific attributes, which can be exploited in order to make the verification feasible.

[*] The author has been supported by grant no. 1ET408050503.
[**] The author has been supported by grant no. 1ET400300504.

Correctness attributes. One of the crucial observations in verification of component interaction in component-based systems is that the correctness attributes often highlight interaction among specific components which form only a small part of the system. Even if the rest of the system is also important as it may coordinate these components, with appropriate reduction techniques a large portion of its complexity could be abstracted away during verification.

Partial-order reduction technique. One of the techniques successfully employed to state-space reduction is the *partial order reduction.* This technique is able to identify redundancies in the model during the verification process, commonly caused by interleaving of independent actions. This allows the technique to omit generation of some of them while at least one representative of each equivalence class remains part of the actually verified model.

State/event temporal logic. In component-based systems, as in any modular programs in general, communication among components proceeds via events, which represent message passing, service calls, delivery of return values, etc. At the same time, components preserve also persistent state information about current values of their attributes. The adequate logic formalizing properties of these systems hence should be able to express both state-based and action-based properties, as well as their combinations. Research conducted on this topic resulted in the state/event LTL [1,2]. For the logic, however, there is no partial order reduction method known at the time. The situation is complicated for its fragment, the action-based LTL, as well.

Contribution. The main contribution of this paper is a partial order reduction method for state/event LTL, and for action-based LTL as its special case. The whole framework is moreover defined in a way that it can be turned at no additional cost into the standard partial order reduction problem for state-based LTL at the end. Hence, it can be resolved with known and widely implemented techniques.

For the reduction to have required effect one needs to identify an equivalence of two runs. The equivalence should allow for considerable level of reduction, while preserving temporal properties that reflect meaningful correctness attributes for the studied systems. We define an equivalence relation, *state/event stuttering equivalence*, driven by the correctness attributes of component-based systems, highlighting only the interesting interaction of components as discussed above, and characterize the state/event LTL properties preserved by the equivalence in terms of a new logic named *weak state/event LTL*.

2 Related Work

A combination of state-based and action-based linear temporal logic, named state/event LTL, has been studied in [1,2]. The authors argue that formalisms including both states and actions are suited for modelling of modular systems, including component-based systems, better than pure state-based or action-based approaches. It is also shown that the automata-based verification method for state-based LTL [3] can be modified to a verification method for state/event LTL in a straightforward way and at no additional cost of time and space. As noted by the authors, the results indicate the importance of further research in reduction techniques. The partial order reduction is suggested as a future direction. Its need was also discovered in our recent work on verification of component-based systems [4,5].

The partial order reduction method was originally introduced in three independent works [6,7,8]. The approach has been further developed, but in connection with linear temporal properties, state-based LTL has always been assumed. The reason for leaving action-based and state/event LTL behind is most likely because the correctness of the partial order reduction method is based on the concept of stuttering invariance of properties [9]. To the best of our knowledge, the stuttering concept has currently no convenient analogue for neither state/event nor action-based LTL. There have been, though, approaches like [10] which solve this problem by transforming state/event systems into purely state-based ones and then using state-based LTL and the standard partial order reduction. However, this comes at a cost, in both enlarging the state space (the number of states can be in worst case multiplied by the size of the alphabet) and weakening the logic (LTL without the next operator is unable to distinguish between one and more consecutive executions of a single event).

An approach that relates actions and stuttering equivalence is the temporal logic of actions [11], where the formulae are constructed in a way that they are stuttering invariant. However, the *actions* are formulated in terms of changes of state propositions and/or variables, not allowing an arbitrary concept of actions. In our approach, we adopt a more general attitude to actions, as we consider arbitrary actions not tied to the properties of states.

The idea behind the state/event stuttering equivalence defined in Section 5 bears many similarities to the concept of *projection* in [12]. However, the logic and the methods studied in [12] are more specialized and quite different from the general case of SE-LTL and partial order reduction.

3 Basic Definitions

As a general modelling formalism for state/event systems we use *labelled Kripke structures*. Many automata-based approaches, such as Component-Interaction Automata [4,5], Interface Automata [13] and I/O automata [14], can be easily translated to labelled Kripke structures.

Definition 1 (LKS). *A* labelled Kripke structure (LKS) *is a 6-tuple* $(S, Act, \Delta, s_{init}, Ap, \mathcal{L})$ *where* S *is a nonempty set of* states*,* Act *is a finite set of* actions*,* $\Delta \subseteq S \times Act \times S$ *is a* transition relation*,* $s_{init} \in S$ *is an* initial state*,* Ap *is a finite set of* atomic propositions *and* $\mathcal{L} \colon S \rightarrow 2^{Ap}$ *is a* state-labelling function*. Instead of* $(s, a, s') \in \Delta$*, we also write* $s \xrightarrow{a} s'$*.*

A run π *of an LKS is an infinite alternating sequence of states and actions* $\pi = s_0, a_0, s_1, a_1, \ldots$ *such that* $\forall i : s_i \xrightarrow{a_i} s_{i+1}$*. We call a run* initial *if* $s_0 = s_{init}$*.*

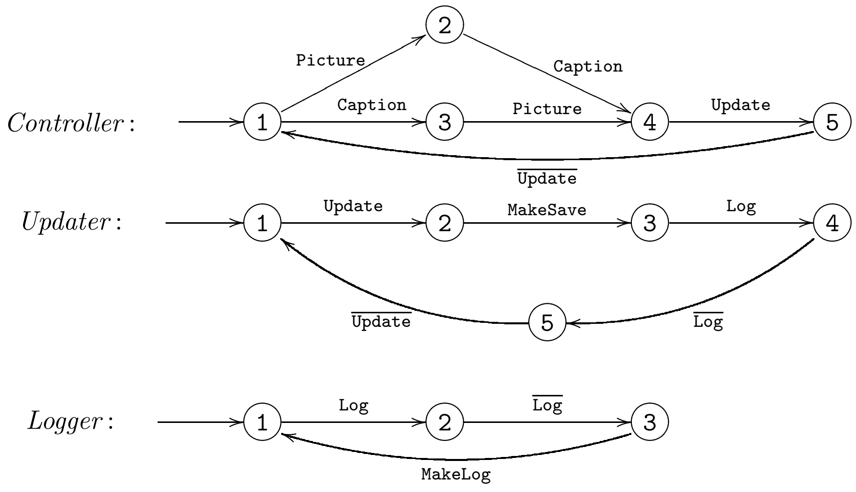


Fig. 1. Models of the *Controller*, *Updater* and *Logger* components

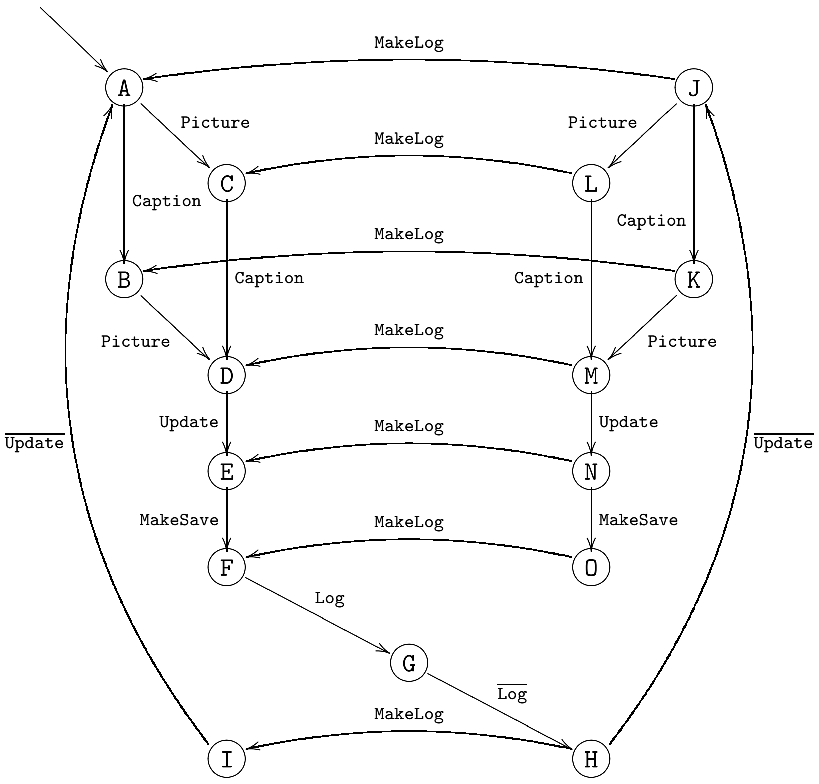


Fig. 2. Model of the *Photo gallery* system

Given a run π, we also define: the ith subrun of π as $\pi^i = s_i, a_i, s_{i+1}, a_{i+1}, \ldots$, the ith state of π as $\pi(i) = s_i$ and the ith action of π as $\ell(\pi, i) = a_i$.

Other kinds of transition systems can be naturally translated into LKSs. The most commonly used are the Kripke structures, which correspond to LKSs without transition labels (i.e. actions), and the labelled transition systems, which correspond to LKSs without atomic propositions and state labelling.

Example 1. Consider a component-based system implementing a simple photo gallery, modelled as an LKS in Figure 2. The system consists of three components: *Controller*, *Updater* and *Logger*, with behavioural descriptions in Figure 1. The interface of the system is formed by *Controller*, which inserts photos with captions into the gallery. When provided with a picture and its caption (in any order), *Controller* asks *Updater* to update information in the gallery via action `Update`, which synchronizes with a corresponding action in *Updater*. In the model of the system in Figure 2, *Updater* gains focus and starts the update. It saves the changes first and then asks *Logger* to log the information. *Logger* responds to the `Log` call via $\overline{\texttt{Log}}$ and then completes the operation. Concurrently with the `MakeLog` operation of *Logger*, *Updater* returns response to *Controller*, which afterall takes all the components (and hence the system) to the initial setting.

To make the LKS model of this system in Figure 2 complete, we need to add the set *Ap* of atomic propositions and the state labelling function $\mathcal{L}$. In this case we choose the atomic propositions reflecting action enabledness. That is, $Ap = \{\mathcal{E}(a) \mid a \in Act\}$ where each $\mathcal{E}(a)$ represents a state proposition with the meaning "action a is enabled in this state". The labelling function is then given as $\mathcal{L}(s) = \{\mathcal{E}(a) \mid \exists s' : s \xrightarrow{a} s'\}$ associating with each state the actions enabled in that state. For example $\mathcal{L}(\texttt{A}) = \{\mathcal{E}(\texttt{Picture}), \mathcal{E}(\texttt{Caption})\}$.

We continue with defining the linear temporal logic that encompasses both state propositions and actions. This definition is equivalent to the definition in [1], it only slightly differs in notation.

Definition 2 (SE-LTL). *Let Act be a set of actions, Ap a set of atomic propositions. The syntax of the* state/event LTL (SE-LTL for short) *formulae is defined inductively as:*

$$\varphi ::= \mathcal{P}(a) \mid p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \,\mathbf{U}\, \varphi_2 \mid \mathbf{X}\,\varphi$$

where a ranges over Act and p ranges over Ap.

Let π be a run of an LKS, the semantics of SE-LTL for runs is defined as:

$$\begin{aligned}
\pi &\models \mathcal{P}(a) && \iff \ell(\pi, 0) = a \\
\pi &\models p && \iff p \in \mathcal{L}(\pi(0)) \\
\pi &\models \neg\varphi && \iff \pi \not\models \varphi \\
\pi &\models \varphi \wedge \psi && \iff \pi \models \varphi \textit{ and } \pi \models \psi \\
\pi &\models \varphi \,\mathbf{U}\, \psi && \iff \exists k \geq 0 : \pi^k \models \psi \textit{ and } \forall j < k : \pi^j \models \varphi \\
\pi &\models \mathbf{X}\,\varphi && \iff \pi^1 \models \varphi
\end{aligned}$$

Further, we say that an LKS M satisfies φ, written as $M \models \varphi$ if for all initial runs π of M, $\pi \models \varphi$.

Example 2. Consider the LKS from Example 1 in Figure 2. Let $\mathbf{F}\,\varphi$ stand for $\mathbf{true}\,\mathbf{U}\,\varphi$ and $\mathbf{G}\,\varphi$ stand for $\neg\,\mathbf{F}\,\neg\varphi$. The property reflecting that an arbitrary number of pictures can be inserted into the album, can be stated in SE-LTL as $\mathbf{G}\,\mathbf{F}\,\mathcal{P}(\texttt{Picture})$. An example of a property using state atomic propositions could be "whenever the action `Picture` becomes enabled, it is eventually executed", which is expressible as $\mathbf{G}\,(\mathcal{E}(\texttt{Picture}) \Rightarrow \mathbf{F}\,\mathcal{P}(\texttt{Picture}))$.

It has been demonstrated in [1] that the automata-based approach for state-based LTL verification (see e.g. [3]) can be straightforwardly transformed into an automata-based verification method for SE-LTL with no extra cost.

4 Motivation

As mentioned in the introduction, to cope with the enormous size of real system models consisting of a large number of components it is necessary to employ reduction methods. The aim of such methods is to generate a reduced state space instead of the complete one while ensuring preservation of all required properties. One of such methods, the partial order reduction method, exploits the redundancies in the system caused by concurrent interleaving. When dealing with state-based LTL properties, the partial order reduction technique is built on the concept of stuttering equivalence [9]. Two runs of a system are considered to be stuttering equivalent if the only difference between them lies in sequential repetitions of states with identical labelling. The partial order reduction method then ensures that for each run of the system there is a stuttering equivalent run in the reduced state space. The subset of LTL properties that are preserved by this equivalence can be characterized syntactically: they are exactly those properties that can be written without the $\mathbf{X}$ operator.

To apply the partial order reduction method to SE-LTL, we need first to find a suitable concept that would play the role of stuttering equivalence for the state/event case. The above mentioned stuttering equivalence cannot be employed, as it considers state labelling only.

The first idea is to transfer the stuttering concept to actions. In the stuttering equivalence, consecutive states labelled with the same atomic propositions are ignored. Let us therefore consider an equivalence that ignores consecutive transitions labelled with the same action and let us call this equivalence *action stuttering*. It implies that, for instance, two runs of the form $s_0, a, s_1, a, s_2, b, s_3, c, s_4, c, \ldots$ and $q_0, a, q_1, b, q_2, b, q_3, b, q_4, c, \ldots$ are action-stuttering equivalent. The advantage of this straightforward approach is that all formulae of action-based LTL not using the $\mathbf{X}$ operator are preserved by this equivalence. However, there is a number of arguments against this choice.

It is obvious that the partial order reduction method does not preserve this action-stuttering equivalence. Figure 3 shows a typical situation. If the transitions labelled with a and b are independent and one of them is invisible, the

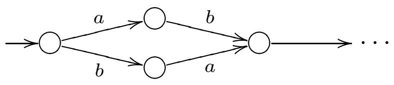


Fig. 3. A typical situation for partial order reduction

partial order reduction method traverses just one of the two runs $ab\ldots$ and $ba\ldots$ However, those two runs are not action-stuttering equivalent.

The problem is more fundamental. Consider a component-based system consisting of two components whose interaction is to be verified. Suppose we extend the system with an additional component that does not influence the communication of the original ones. A suitable substitution for stuttering equivalence should consider every run corresponding to the interaction behaviour of the original components equivalent regardless of interleaving with the third component. It is clear that the proposed action-stuttering equivalence does not satisfy this reasonable property.

We define a new equivalence, which, while still retaining the stuttering concept with respect to the state propositions, employs a different approach towards the transition labels (actions). This new equivalence enjoys the property that it is preserved by the partial order reduction method, thus allowing all the advantages of it. This comes at a cost. Contrary to state-based LTL, we do not have any syntactic characterization of SE-LTL formulae that are preserved by the new equivalence. However, we show that they can be elegantly described in terms of an adjusted weak version of SE-LTL.

5 State/Event Stuttering Equivalence

The main idea of the equivalence is that some of the actions are regarded as *interesting*. Transitions with noninteresting actions are then overlooked by the equivalence. As we want to consider both actions and states, this idea is combined with the stuttering principle for state propositions, i.e. transitions which change state propositions we are interested in cannot be overlooked. In order to define the equivalence formally, we introduce the notions of a projection and a signature.

Definition 3 (projection, signature). *Let $\pi = s_0, a_0, s_1, a_1, \ldots$ be a run of LKS $(S, Act, \Delta, s_{init}, Ap, \mathcal{L})$, let $Act' \subseteq Act$ and $Ap' \subseteq Ap$. Let τ be a new symbol, $\tau \notin Act$. A* projection of π onto Act' and Ap' *is defined as*

$$\mathsf{pr}^{Ap'}_{Act'}(\pi) = E_0, b_0, E_1, b_1, \ldots$$

where $E_i = \mathcal{L}(s_i) \cap Ap'$, b_i is equal to τ whenever $a_i \notin Act'$ and $b_i = a_i$ otherwise.

Furthermore, a signature of π with respect to Act' and Ap', *denoted as* $\mathsf{sig}^{Ap'}_{Act'}(\pi)$ *is defined as the (finite or infinite) alternating sequence of sets of atomic propositions and actions that arises from the projection of π onto Act' and Ap' by replacing every maximal subsequence of the form $E_i, \tau, E_{i+1}, \tau, \ldots$, where $E_i = E_{i+1} = \cdots$, with just E_i.*

Example 3. Let $\pi = \mathtt{A}, \mathtt{Caption}, \mathtt{B}, \mathtt{Picture}, \mathtt{D}, \mathtt{Update}, \mathtt{E}, \mathtt{MakeSave}, \mathtt{F}, \dots$ and let $Act' = \{\mathtt{Update}\}$ and $Ap' = \{\mathcal{E}(\mathtt{Picture})\}$. Then the projection of π onto Act' and Ap' is $\mathsf{pr}_{Act'}^{Ap'}(\pi) = \{\mathcal{E}(\mathtt{Picture})\}, \tau, \{\mathcal{E}(\mathtt{Picture})\}, \tau, \emptyset, \mathtt{Update}, \emptyset, \tau, \emptyset, \dots$ and its signature is $\mathsf{sig}_{Act'}^{Ap'}(\pi) = \{\mathcal{E}(\mathtt{Picture})\}, \tau, \emptyset, \mathtt{Update}, \emptyset, \dots$

Definition 4 (state/event stuttering equivalence). *Let π and σ be two runs, let Act' be a set of actions, Ap' a set of atomic propositions. We say that π and σ are* state/event stuttering equivalent with respect to Act' and Ap', *denoted as $\pi \equiv_{Act'}^{Ap'} \sigma$, if they have the same signatures, i.e. $\mathsf{sig}_{Act'}^{Ap'}(\pi) = \mathsf{sig}_{Act'}^{Ap'}(\sigma)$.*

Two LKSs are said to be state/event stuttering equivalent with respect to Act' and Ap', if for each run of one LKS there is a state/event stuttering equivalent run of the other and vice versa.

Stuttering equivalence is a special case of state/event stuttering equivalence for $Act' = \emptyset$.

Definition 5 (weak SE-LTL). *Let Act be a set of actions, Ap a set of atomic propositions and let $Act' \subseteq Act$. We define the* weak state/event LTL with respect to Act', *wSE-LTL for short, as follows. The syntax of the formulae is defined inductively as:*

$$\varphi ::= \widetilde{\mathcal{P}}(a) \mid p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \,\mathbf{U}\, \varphi_2 \mid \widetilde{\mathbf{X}}\, \varphi \mid \varphi_1 \,\mathbf{U}_a\, \varphi_2$$

where a ranges over Act' and p ranges over Ap.

Let π be a run of an LKS. The semantics for runs is defined inductively as

$$\begin{array}{lll}
\pi \models \widetilde{\mathcal{P}}(a) & \iff & \exists k \geq 0 : \ell(\pi, k) = a \text{ and } \forall j < k : \ell(\pi, j) \notin Act' \\
\pi \models p & \iff & p \in \mathcal{L}(\pi(0)) \\
\pi \models \neg\varphi & \iff & \pi \not\models \varphi \\
\pi \models \varphi \wedge \psi & \iff & \pi \models \varphi \text{ and } \pi \models \psi \\
\pi \models \varphi \,\mathbf{U}\, \psi & \iff & \exists k \geq 0 : \pi^k \models \psi \text{ and } \forall j < k : \pi^j \models \varphi \\
\pi \models \widetilde{\mathbf{X}}\, \varphi & \iff & \exists k \geq 0 : \ell(\pi, k) \in Act', \forall j < k : \ell(\pi, j) \notin Act' \text{ and } \pi^{k+1} \models \varphi \\
\pi \models \varphi \,\mathbf{U}_a\, \psi & \iff & \exists k \geq 0 : \ell(\pi, k) = a, \pi^{k+1} \models \psi \text{ and } \forall j < k+1 : \pi^j \models \varphi
\end{array}$$

The main difference between SE-LTL and wSE-LTL is in the semantics of the next and action operators. While $\mathcal{P}(a)$ states that "the first action is a", $\widetilde{\mathcal{P}}(a)$ states that "the first *interesting* action is a". The formula $\mathbf{X}\, \varphi$ states that "in the next step, φ holds", while the formula $\widetilde{\mathbf{X}}\, \varphi$ states that "after the next *interesting action*, φ holds". Actually, the definition of $\widetilde{\mathbf{X}}\, \varphi$ is of the form "there is a next interesting action *and* after this action, φ holds". We could also sometimes be interested in stating the property that "*if* there is a next interesting action, *then* after this action, φ holds". Nevertheless, this alternative $\widehat{\mathbf{X}}\, \varphi$ operator can be defined as a derived operator: $\widehat{\mathbf{X}}\, \varphi := \neg\, \widetilde{\mathbf{X}}\, \neg\varphi$.

Additionally, wSE-LTL has a new operator $\mathbf{U}_a$. The motivation is to express properties like "atomic proposition p holds until action a happens". In SE-LTL, this can be expressed with $p\,\mathbf{U}\,(p \wedge \mathcal{P}(a))$. In wSE-LTL this is no longer possible as the semantics of $\widetilde{\mathcal{P}}(a)$ is different and the intuitive solution $p\mathbf{U}\,(p\wedge\widetilde{\mathcal{P}}(a))$ holds even for runs that do not satisfy the original property, e.g. a run with signature $\{p\}, \tau, \{\neg p\}, a, \ldots$ Thanks to the $\mathbf{U}_a$ operator, this property is expressible as $p\,\mathbf{U}_a\,\mathbf{true}$. The $\mathbf{U}_a$ operator is not needed in the action-based fragment of wSE-LTL. The reader may verify that every formula $\varphi\,\mathbf{U}_a\,\psi$, where φ and ψ both do not use state atomic propositions, is equivalent to $\varphi\,\mathbf{U}\,(\widetilde{\mathcal{P}}(a) \wedge \varphi \wedge \widetilde{\mathbf{X}}\,\psi)$.

In the previous, *interesting* means "from Act'". In many natural cases, the interpretation of a wSE-LTL formula remains the same regardless of the choice of Act' as long as Act' contains all action labels from $\mathcal{P}(a)$ and $\mathbf{U}_a$ subformulae of the formula. This is, however, not true in general, as semantics of some wSE-LTL formulae may depend on this choice. When specifying properties in wSE-LTL, it is therefore assumed that a pair (φ, Act') is given instead of just φ.

An example of a formula that has different validity depending on the choice of Act' is given in the following.

Example 4. Consider the LKS from Example 1 in Figure 2 and the formula

$$\mathbf{G}(\widetilde{\mathcal{P}}(\overline{\texttt{Update}}) \Rightarrow \widetilde{\mathbf{X}}((\mathcal{E}(\texttt{Picture}) \vee \mathcal{E}(\texttt{Caption}))\,\mathbf{U}\,\widetilde{\mathcal{P}}(\texttt{MakeSave})))$$

expressing the property that "After finishing the update, it is possible to add a picture or a caption until `MakeSave` follows as the next observable action." This formula holds if Act' is given as $\{\overline{\texttt{Update}}, \texttt{MakeSave}\}$. However, it does not hold when the action label `Update` is added to Act'.

We are now ready to present the main result of this section, namely that the properties expressible in weak SE-LTL are preserved by the state/event stuttering equivalence. The proof of the following theorem can be found in [15].

Theorem 1 (equivalence preserves properties). *Let Act' be a set of actions, Ap' a set of atomic propositions, and let π and σ be two runs such that $\pi \equiv^{Ap'}_{Act'} \sigma$. Then for each formula φ of wSE-LTL with respect to Act' such that it only contains atomic propositions from Ap', $\pi \models \varphi$ if and only if $\sigma \models \varphi$.*

What remains is to show that wSE-LTL is indeed a weak version of SE-LTL, i.e. that all properties expressible in wSE-LTL are also expressible in SE-LTL. The following theorem states that every formula of wSE-LTL can be translated in linear time to an equivalent formula of SE-LTL. This way the verification problem for wSE-LTL can be reduced to the verification problem for SE-LTL, which is solvable in a way similar to the standard LTL verification as described in [1].

Theorem 2 (embedding of wSE-LTL into SE-LTL). *Every formula φ of weak SE-LTL with respect to Act' can be translated to a formula $T(\varphi)$ of SE-LTL such that for each π, $\pi \models \varphi$ if and only if $\pi \models T(\varphi)$.*

Proof. We define an auxiliary formula $\xi := \bigwedge_{a \in Act'} \neg \mathcal{P}(a)$. The translation is defined inductively as follows:

$$\begin{aligned}
T(p) &:= p \\
T(\widetilde{\mathcal{P}}(a)) &:= \xi \,\mathbf{U}\, \mathcal{P}(a) \\
T(\widetilde{\mathbf{X}}\, \varphi) &:= \xi \,\mathbf{U}\, (\neg \xi \wedge \mathbf{X}\, T(\varphi)) \\
\\
T(\varphi \,\mathbf{U}\, \psi) &:= T(\varphi) \,\mathbf{U}\, T(\psi) \\
T(\varphi \,\mathbf{U}_a\, \psi) &:= T(\varphi) \,\mathbf{U}\, (\mathcal{P}(a) \wedge T(\varphi) \wedge \mathbf{X}\, T(\psi)) \\
T(\varphi \wedge \psi) &:= T(\varphi) \wedge T(\psi) \\
T(\neg \varphi) &:= \neg T(\varphi)
\end{aligned}$$

The correctness of this construction is proved in [15]. □

5.1 Characterization of Invariant SE-LTL Properties

We have shown that wSE-LTL is preserved by state/event stuttering equivalence and can be embedded into SE-LTL. Thus, wSE-LTL can be seen as a characterization of some SE-LTL properties that are preserved by state/event stuttering equivalence (we use the term *state/event stutter-invariant* for such properties in the following). We now show that this characterization is exact, i.e. that all state/event stutter-invariant SE-LTL properties are expressible in wSE-LTL. The proof follows the method of [9].

Definition 6. *A run π is* state/event stutter-free, *if for each $i \geq 0$ one of the following holds:*

- $\ell(\pi, i) \in Act'$ *(ith transition is labelled by interesting action)*
- $\mathcal{L}(\pi(i)) \cap Ap' \neq \mathcal{L}(\pi(i+1)) \cap Ap'$ *(ith transition changes the state labelling)*
- $\ell(\pi, j) \notin Act'$ *and* $\mathcal{L}(\pi(j)) \cap Ap' = \mathcal{L}(\pi(j+1)) \cap Ap'$ *for all $j \geq i$ (nothing interesting ever happens from ith position onwards)*

It is clear that a state/event stutter-free run is a unique representant of its state/event stuttering equivalence class. Note that an arbitrary subrun of a state/event stutter-free run is also state/event stutter-free.

Theorem 3. *Every state/event stutter-invariant property expressible in SE-LTL is expressible in wSE-LTL.*

Proof. We will show that for every SE-LTL formula φ there exists a wSE-LTL formula $\tau(\varphi)$ that agrees with φ on all state/event stutter-free runs. Clearly this implies the theorem.

The formula $\tau(\varphi)$ is defined inductively as follows. The straightforward parts are $\tau(p) = p$ for $p \in Ap'$, $\tau(\neg\varphi) = \neg\tau(\varphi)$, $\tau(\varphi \wedge \psi) = \tau(\varphi) \wedge \tau(\psi)$ and $\tau(\varphi \,\mathbf{U}\, \psi) = \tau(\varphi) \,\mathbf{U}\, \tau(\psi)$. These choices are obviously correct. The more difficult parts are that of $\mathcal{P}(a)$ and $\mathbf{X}\, \varphi$.

Assume that $Ap' = \{p_1, \dots, p_n\}$ and let N be the set of all subsets of Ap', i.e. $N = 2^{Ap'}$. For each $\nu \in N$, let β_ν be the formula $\alpha_1 \wedge \dots \wedge \alpha_n$ where $\alpha_j = p_j$ if $p_j \in \nu$ and $\alpha_j = \neg p_j$ otherwise. If $Ap' = \emptyset$ then $N = \{\emptyset\}$ and $\beta_\emptyset = \mathbf{true}$. Thus, β_ν holds in precisely those states whose valuation is equal to ν.

The two remaining formulae can be then dealt with as follows:

$$\tau(\mathcal{P}(a)) := \widetilde{\mathcal{P}}(a) \wedge \bigvee_{\nu \in N} \beta_\nu \, \mathbf{U}_a \, \mathbf{true}$$

$$\tau(\mathbf{X}\,\varphi) := \psi_1 \vee \psi_2 \vee \psi_3, \text{ where}$$

$$\psi_1 := \bigvee_{\nu \in N} \Big(\mathbf{G}\,\beta_\nu \wedge \neg\, \widetilde{\mathbf{X}}\,\mathbf{true} \wedge \tau(\varphi) \Big)$$

$$\psi_2 := \bigvee_{\nu \in N} \bigvee_{a \in Act'} \Big(\beta_\nu \, \mathbf{U}_a \, \mathbf{true} \wedge \widetilde{\mathbf{X}}\,\tau(\varphi) \Big)$$

$$\psi_3 := \neg \bigvee_{\nu \in N} \bigvee_{a \in Act'} (\beta_\nu \, \mathbf{U}_a \, \mathbf{true}) \wedge \bigvee_{\nu \in N} \bigvee_{\nu' \in N \smallsetminus \{\nu\}} (\beta_\nu \wedge (\beta_\nu \, \mathbf{U} \, (\beta_{\nu'} \wedge \tau(\varphi))))$$

The correctness of this choice is proved in [15]. □

Although the construction in the above proof is exponential in worst case, the main significance of this result is that using weak SE-LTL comes without loss of expressiveness over SE-LTL if we want to use state/event stutter-invariant properties. Moreover, the construction justifies the choice of wSE-LTL operators, namely the $\widetilde{\mathcal{P}}$, $\widetilde{\mathbf{X}}$ and $\mathbf{U}_a$ operators, which made the construction possible. Note that if any of these were excluded from the logic, it would be less expressive and the above result would not hold.

6 Partial Order Reduction

The goal of this section is to show that the partial order reduction method can be applied to LKSs so that the reduced LKS remains state/event stuttering equivalent. At first, we summarize the basics of the partial order reduction method. While presenting the method we follow the explication from [16]. Consequently, we explain how the method can be applied to SE-LTL.

Definition 7 (state transition system). *A* state transition system *is a triple (S, T, s_{init}) where S is a set of states, s_{init} is an initial state and T is a set of transitions such that for each $\alpha \in T$, $\alpha \subseteq S \times S$. Further, for each $\alpha \in T$ and for each state $s \in S$ there is at most one $s' \in S$ such that $(s, s') \in \alpha$.*

An initial transition path *of a state transition system is an infinite sequence $\alpha_0\alpha_1 \dots$ such that there are states $s_0, s_1, \dots$ satisfying $s_0 = s_{init}$ and for all i, $(s_i, s_{i+1}) \in \alpha_i$.*

The idea of the ample set method [7,17] is to construct a reduced state space by choosing a smaller set of successors at each state. Instead of exploring all successors from a given state, denoted as *enabled*(s), we explore only successors from *ample*$(s) \subseteq$ *enabled*(s).

Theorem 4. *[16] Let M be a state transition system and let $V \subseteq T$ be an arbitrary set of* visible *transitions. Let M' be the reduced system constructed using the ample set partial order algorithm. Then for each initial transition path π from M there is an initial transition path σ in M' such that π and σ have the same sequence of visible transitions.*

Proof. The theorem follows from the proof of Theorem 12 in [16]. □

Now we present a transformation of an LKS to a state transition system. From now on, we fix the LKS $\mathcal{K} = (S, Act, \Delta, s_{init}, Ap, \mathcal{L})$, and two sets, $Act' \subseteq Act$ and $Ap' \subseteq Ap$ of interesting actions and interesting atomic propositions respectively. We need the notions of invisibility and proper transition partition first.

Definition 8 (invisibility). *A transition $(q, a, r) \in \Delta$ is called* invisible *if $a \notin Act'$ and $\mathcal{L}(q) \cap Ap' = \mathcal{L}(r) \cap Ap'$. A transition is called* visible *if it is not invisible.*

Definition 9 (proper transition partition). *An indexed set $P = \{\Delta_i \mid i \in I\}$ is called a* proper transition partition *if the following holds:*

- *P is a partition of Δ, i.e. $\bigcup_{i \in I} \Delta_i = \Delta$ and $\Delta_i \cap \Delta_j = \emptyset$ for all $i \neq j$.*
- *P preserves actions, i.e. $\forall i \in I \,.\, \exists a \in Act : \Delta_i \subseteq S \times \{a\} \times S$.*
- *P is deterministic, i.e. for all i and for all $s \in S$, there is at most one $s' \in S$ such that $(s, a, s') \in \Delta_i$.*

We now present a transformation of LKS $\mathcal{K} = (S, Act, \Delta, s_{init}, Ap, \mathcal{L})$ with a proper transition partition P into a state transition system $\mathcal{M} = (S, T, s_{init})$. Let $T = \{\alpha_i \mid i \in I\}$ and $\alpha_i = \{(s, s') \mid (s, a, s') \in \Delta_i\}$. The set of visible transitions $V \subseteq T$ is defined as $V = \{\alpha_i \mid \text{at least one transition in } \Delta_i \text{ is visible}\}$. We denote $\Delta(\alpha_i) = \Delta_i$ the underlying partition set for α_i. Let $\alpha = \alpha_0\alpha_1 \ldots$ be an initial transition path of the state transition system (S, T, s_{init}). We assign to α an initial run $s_0, a_0, s_1, a_1, \ldots$ where $s_0 = s_{init}$, $(s_i, a_i, s_{i+1}) \in \Delta(\alpha_i)$ for $i \geq 0$. Clearly, the assigned initial run is a unique initial run of the LKS due to the nature of the proper transition partition.

Theorem 5. *Let $\alpha = \alpha_0\alpha_1 \ldots$ and $\beta = \beta_0\beta_1 \ldots$ be two initial transition paths of the state transition system (S, T, s_{init}). Let π and σ be the initial runs assigned to α and β, respectively. If α and β have the same sequence of visible transitions, then $\pi \equiv_{Act'}^{Ap'} \sigma$.*

Proof. Invisible transitions of the state transition system represent transitions of the original LKS such that they do not contribute to the runs' signatures. □

Corollary 1. *Let M be an LKS and let $Act' \subseteq Act$ and $Ap' \subseteq Ap$. Let M' be the reduced system constructed using the ample set partial order algorithm on the state transition system that is created as discussed above. Then for each initial run π of M there is an initial run σ of M' such that $\pi \equiv_{Act'}^{Ap'} \sigma$.*

Three things have to be supplied to the partial order reduction algorithm along with the LKS to be verified in order for the method to work. They are the proper

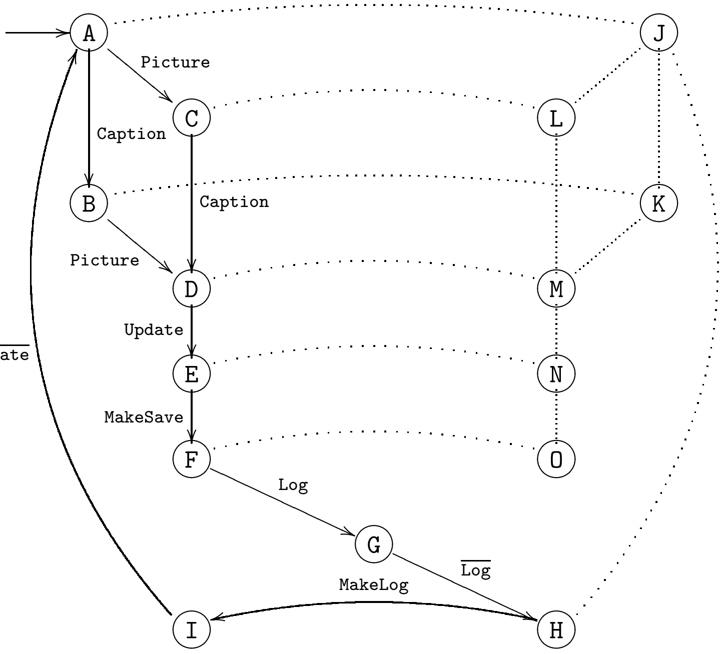


Fig. 4. Example from Figure 2 after partial order reduction

transition partition, the set of interesting atomic propositions Ap' and the set of interesting actions Act'.

A proper transition partition can be constructed automatically from some additional information of the structure of the LKS in question. Take for instance that the LKS represents a component-based system made from a number of smaller components such as the LKS in Example 1. A proper transition partition can be constructed in such a way that all transitions of the system that correspond with one transition of a smaller component constitute exactly one set of the partition.

A set of interesting atomic propositions is acquired from the verified formula. It is constructed as the set of all atomic propositions present in the formula. A set of interesting actions, however, has to be supplied by the user by hand. This set has to be a part of the property specification, as noted in Section 5, nonetheless. The cardinality of this set can affect the effectivity of the reduction method. It is thus desirable to specify as small set of interesting actions as possible, bearing in mind the intended semantics of the verified formula.

The partial order reduction itself can then be done on the fly during the automata-based verification process. Known methods and implementations can be used for this purpose, see e.g. [18].

Example 5. Consider the LKS from Example 1. If we choose Ap' to be an arbitrary subset of Ap and Act' to be an arbitrary subset of Act such that $\texttt{MakeLog} \notin Act'$ then the state space of the reduced LKS will look as depicted in Figure 4. In this case $ample(s) = enabled(s)$ for all states except H and $ample(\texttt{H}) = \{\texttt{I}\}$.

7 Conclusion and Future Work

The paper introduces a partial order reduction technique for state/event LTL. The technique is based on a new stuttering equivalence, which is able to reflect both state and transition labels while regarding both with a different principle to closely fit their nature. On the level of states, the stuttering concentrates on changes in assigned atomic propositions along a run, whereas in the case of actions, the interesting events are observed at every single occurrence of an action representing interesting behaviour of the system, which is stated explicitly with respect to the verified property for instance. The paper moreover gives the characterization of the state/event LTL properties preserved by the equivalence, and summarizes the attributes into the definition of its fragment, called weak state/event LTL. This fragment is preserved by the equivalence while staying strong enough to reflect interesting component-specific properties, discussed at the beginning of this paper.

After introducing both the equivalence and the corresponding logic fragment, we discuss the partial order reduction technique. We show that the partial order reduction task for the state/event case can be translated into the state-based case via providing existing algorithms with a modified definition of transition invisibility reflecting the discussed specifics. The advantage of such translation is that known algorithms may be used for solving the problem. Moreover, the efficiency of the partial order reduction method, which has been experimentally verified on a number of case studies [18,19,20], can be attributed to our approach as well.

Future work. Our ongoing and future aims are connected to the employment of the technique into our framework for formal verification of component-based systems [21] based upon the formalism of Component-Interaction automata [4,22]. The framework, in connection with the model checking tool DiVinE [23], recently helped us to perform an extensive verification case study [5], which uncovered the need of a partial order reduction method for state/event systems not found elsewhere. Currently we work on the implementation and plan to perform a detailed experimental case study using our framework and Component-Interaction automata as an underlying formalism. The technique, however, is general and independent on the application we aim it for.

References

1. Chaki, S., Clarke, E.M., Ouaknine, J., Sharygina, N., Sinha, N.: State/event-based software model checking. In: Boiten, E.A., Derrick, J., Smith, G.P. (eds.) IFM 2004. LNCS, vol. 2999, pp. 128–147. Springer, Heidelberg (2004)
2. Chaki, S., Clarke, E., Ouaknine, J., Sharygina, N., Sinha, N.: Concurrent software verification with states, events, and deadlocks. Formal Aspects of Computing 17(4), 461–483 (2005)
3. Gerth, R., Peled, D., Vardi, M.Y., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: Proceedings of PSTV 1995, Warsaw, Poland, pp. 3–18. Chapman and Hall, Boca Raton (1995)

4. Zimmerova, B., Vařeková, P., Beneš, N., Černá, I., Brim, L., Sochor, J.: Component-Interaction Automata Approach (CoIn). In: Rausch, A., Reussner, R., Mirandola, R., Plášil, F. (eds.) The Common Component Modeling Example. LNCS, vol. 5153, pp. 146–176. Springer, Heidelberg (2008)
5. Beneš, N., Černá, I., Sochor, J., Vařeková, P., Zimmerova, B.: A case study in parallel verification of component-based systems. In: Proceedings of PDMC 2008, pp. 35–51 (2008)
6. Valmari, A.: A stubborn attack on state explosion. In: Clarke, E., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 156–165. Springer, Heidelberg (1991)
7. Peled, D.: All from one, one from all: on model checking using representatives. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 409–423. Springer, Heidelberg (1993)
8. Godefroid, P.: Using partial orders to improve automatic verification methods. In: Clarke, E., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 176–185. Springer, Heidelberg (1991)
9. Peled, D., Wilke, T.: Stutter-invariant temporal properties are expressible without the next-time operator. Information Processing Letters 63(5), 243–246 (1997)
10. Sun, J., Liu, Y., Dong, J.S., Wang, H.H.: Specifying and verifying event-based fairness enhanced systems. In: Liu, S., Maibaum, T., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 5–24. Springer, Heidelberg (2008)
11. Lamport, L.: The temporal logic of actions. ACM Transactions on Programming Languages and Systems 16(3), 872–923 (1994)
12. Brückner, I., Wehrheim, H.: Slicing object-Z specifications for verification. In: Treharne, H., King, S., Henson, M.C., Schneider, S. (eds.) ZB 2005. LNCS, vol. 3455, pp. 414–433. Springer, Heidelberg (2005)
13. de Alfaro, L., Henzinger, T.A.: Interface-based design. In: Proceedings of the 2004 Marktoberdorf Summer School. Kluwer, The Netherlands (2005)
14. Lynch, N.A., Tuttle, M.R.: An introduction to input/output automata. CWI Quarterly 2(3), 219–246 (1989)
15. Beneš, N., Brim, L., Černá, I., Sochor, J., Vařeková, P., Zimmerova, B.: Partial Order Reduction for State/Event LTL. Technical Report FIMU-RS-2008-07, Masaryk University, Faculty of Informatics, Brno, Czech Republic (July 2008), `http://www.fi.muni.cz/reports/files/2008/FIMU-RS-2008-07.pdf`
16. Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press, Cambridge (1999)
17. Peled, D.: Combining partial order reductions with on-the-fly model-checking. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 377–390. Springer, Heidelberg (1994)
18. Bosnacki, D., Leue, S., Lluch-Lafuente, A.: Partial-order reduction for general state exploring algorithms. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 271–287. Springer, Heidelberg (2006)
19. Peled, D.: Ten years of partial order reduction. In: Y. Vardi, M. (ed.) CAV 1998. LNCS, vol. 1427, pp. 17–28. Springer, Heidelberg (1998)
20. Holzmann, G.J., Peled, D.: An improvement in formal verification. In: FORTE, London, UK, pp. 197–211. Chapman and Hall, Boca Raton (1994)
21. The CoIn Team: CoIn Tool – Formal Verification Tool for Component Interaction Automata, `http://anna.fi.muni.cz/coin/tool/`
22. Brim, L., Černá, I., Vařeková, P., Zimmerova, B.: Component-Interaction automata as a verification-oriented component-based system specification. In: Proceedings of SAVCBS 2005, pp. 31–38. ACM Press, New York (2005)
23. Barnat, J., Brim, L., Černá, I., Moravec, P., Ročkai, P., Šimeček, P.: DiVinE – a tool for distributed verification. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 278–281. Springer, Heidelberg (2006)

Dynamic Path Reduction for Software Model Checking*

Zijiang Yang[1], Bashar Al-Rawi[2], Karem Sakallah[2], Xiaowan Huang[3], Scott Smolka[3], and Radu Grosu[3]

[1] Western Michigan University, Kalamazoo, MI, USA
[2] University of Michigan, Ann Arbor, MI, USA
[3] Stony Brook University, Stony Book, NY, USA

Abstract. We present the technique of *dynamic path reduction* (DPR), which allows one to prune redundant paths from the state space of a program under verification. DPR is based on the symbolic analysis of concrete executions. For each explored execution path π that does not reach an `abort` statement, we repeatedly apply a weakest-precondition computation to accumulate the constraints associated with an infeasible sub-path derived from π by taking the alternative branch to an if-then-else statement. We then use an SMT solver to learn the minimally unsatisfiable core of these constraints. By further learning the statements in π that are critical to the sub-path's infeasibility as well as the control-flow decisions that must be taken to execute these statements, unexplored paths containing the same unsatisfiable core can be efficiently and dynamically pruned. DPR is a very general technique which we consider here in the context of the bounded model checking of sequential programs with nondeterministic conditionals. Our preliminary experimental results show that DPR can prune a significant percentage of execution paths, a percentage that grows with the size of the instance of the problem being considered.

1 Introduction

There are two approaches to software model checking. The first, as typified by [1], applies traditional model-checking techniques [10] to a finite-state model automatically extracted from the software system in question. The use of abstraction techniques [3] leads to a model with more behaviors than the original program and consequently an analysis that is conservative in nature. This form of software model checking allows one to prove the absence of certain types of errors without actually executing the program. Its success hinges on recent advances in symbolic techniques. Performance can be further improved by exploiting software-specific features [15,16].

The second approach is based on the dynamic execution of the actual program; see, for example, [6]. It differs from testing in that it explores exhaustively

* This work was partially supported by NSF grants CCF-0811287 and CCF-0810865.

the program's state space. This approach allows for the detection of subtle implementation errors that are usually missed by abstraction-based software model checkers. In the case of concurrent programs, partial-order reduction [13,11,5] can be used to reduce the size of the state space by exploiting the commutativity of concurrently executed transitions that result in the same state when executed in different orders. However, there is no general technique to reduce the state space in dynamic execution based model checking of sequential programs.

In this paper, we present the technique of *dynamic path reduction* (DPR), which allows one to prune redundant paths from the state space of a program in dynamic execution based model checking. DPR is a very general technique which we consider here in the context of the bounded model checking of sequential programs with nondeterministic conditionals. Such programs arise naturally as a byproduct of abstraction during verification as well as being inherent in nondeterministic programming languages. Nondeterministic choice also arises in the modeling of randomized algorithms. The key idea behind DPR is to *learn* from each explored path so that unexplored paths exhibiting the same behavior can be avoided.

To illustrate the DPR approach to model checking, consider the C program of Figure 1(a). Its first two conditional statements are nondeterministic, denoted by placing an asterisk in the condition position. The property we would like to check is whether the program can reach the `abort` statement. The initial values of variables `x, y, z` are 5, 8, 20, respectively. Suppose the first executed path is $\pi = \langle 0, 1, 2, 5, 6, 7, 10, 13 \rangle$. Executing the program along this path avoids the `abort` statement and ends with the `halt` statement. After executing this path, the existing dynamic execution based model checkers will backtrack to Line 6 and explore the else-branch in the next execution. Since there are two nondeterministic choices in the program, four executions are required to prove that it cannot be aborted.

This is where DPR can be applied. Analyzing the execution trace π allows us to learn that the assignments `x = 5` and `x = 2*x` falsify the predicate `x>10` which forces the third conditional to choose its else-branch. We also learn that none of the assignments within the branches of the nondeterministic conditionals can make the predicate true. This allows us to prune all the remaining paths from the search space. A DPR-based model checker would therefore stop after the first execution and report that `abort` is not reachable.

The rest of the paper is organized as follows. Section 2 presents our execution-based, bounded model-checking algorithm with dynamic path reduction. Section 3 discusses our space-efficient, symbolic representation of execution paths. Section 4 contains our experimental results, while Section 5 considers related work. Section 6 offers our concluding remarks and directions for future work.

2 DPR-Based Model Checking Algorithm

In this section, we present `DPR-MC`, our bounded model-checking algorithm with dynamic path reduction. Our presentation is carried out in stages, starting with

```
foo( ) {
    x=5,y=8,z=20;
    if (*)
        y = y-1;
    else
        y = y+1;
    x = 2*x;
    if (*)
        z = z-1;
    else
        z = z+1;
    if (x>10)
        abort;
    else
        halt;
}
```

```
foo_SSA( ){
    x1=5,y1=8,z1=20;
    if (*)
        y2 = y1-1;
    else
        y3 = y1+1;
    y4 = φ(y2,y3);
    x2 = 2*x1;
    if (*)
        z2 = z1-1;
    else
        z3 = z1+1;
    z4 = φ(z2,z3);
    if (x2 >10)
        abort;
    else
        halt;
}
```

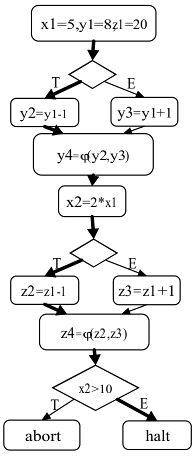


Fig. 1. A sample C program (left), its SSA form (middle), and SSA graph representation (right)

a simplified but transparent version of the algorithm, and with each stage incrementally improving the algorithm's performance. The model-checking algorithm we propose is tunable to run either as a randomized Las Vegas algorithm or as a guided-search algorithm.

As defined formally below, a k-oracle is a bit string of length k representing a sequence of nondeterministic choices a program might make during execution. Suppose we want to perform bounded model checking on a program up to search depth D, such that within this D-bounded search space, each execution path contains at most k nondeterministic choices. In this case, the DPR-MC algorithm repeats the following three steps until the entire D-bounded search space has been explored: (1) Ask a constraint solver to provide a k-oracle. (2) Execute the program on that oracle; stop if an abort statement is reached. (3) Use a constraint solver to prune from the search space all paths that are equivalent to the one just executed.

2.1 Global Search Algorithm

The core language we use for analysis is a subset of C, extended with one statement type not present in C: nondeterministic conditionals. To simplify the analysis undertaken by DPR-MC, we use the *static single assignment* (SSA) representation of programs. For example, the SSA representation of the C program of Figure 1 (left) is shown in Figure 1 (middle). By indexing (versioning) variables and introducing the so-called φ function at join points, this intermediate representation ensures that each variable is statically assigned only once. We leverage the SSA representation to interface with the satisfiability modulo theory (SMT) solver Yices [4]. In this context, every statement (excepting statements within loops) can be conveniently represented as a predicate. Looping statements are handled by unfolding them up so that every execution path has at most k nondeterministic choices; i.e., a k-oracle is used to resolve the choices. We refer to the SSA representation obtained after such a k-unfolding as the *dynamic single assignment* (DSA) representation.

Suppose the program C to be analyzed has at most k nondeterministic conditionals on every execution path. We call a resolution of these k conditionals a *k-oracle*. Obviously, each k-oracle uniquely determines a finite concrete execution path of C. Let $\mathcal{R}$ be the set of all k-oracles (resolvents) of C. $\mathcal{R}$ can be organized as a decision tree whose paths are k-oracles.

Algorithm 1. DPR-MC(PROGRAM C, INT k)

```
1: $\mathcal{R}$ = all $k$-oracles in $C$;
2: while $\mathcal{R} \neq \emptyset$ do
3:     Remove an oracle $R = \langle r_1 r_2 \ldots r_k \rangle$ from $\mathcal{R}$;
4:     ExecuteFollowOracle($R, \mathcal{R}, k$);
5: end while
6: exit(“No bug found up to oracle-depth $k$”);
```

Algorithm 1 is the main loop of our `DPR-MC` algorithm. It repeatedly removes a k-oracle R from $\mathcal{R}$ and executes C as guided by R. The algorithm terminates if: (1) execution reaches `abort` within `ExecuteFollowOracle`, indicating that a bug is found; or (2) $\mathcal{R}$ becomes empty after all oracles have been explored, in which case the program is bug-free to oracle-depth k.

Note that Algorithm 1 employs a *global* search strategy. If the oracle removal is random, it corresponds to a randomized Las Vegas algorithm. If the oracle removal is heuristic, it corresponds to a guided-search algorithm. Obviously, the number of oracles is exponential in the depth k of the decision tree $\mathcal{R}$. Hence, the algorithm is unlikely to work for nontrivial programs. We subsequently shall show how to efficiently store the decision tree and how to prune oracles by learning from previous executions.

2.2 Weakest Precondition Computation

An execution path $\pi = \langle s_1, s_2, \ldots, s_n \rangle$ is a sequence of program statements, where each s_i is either an assignment or a conditional. We write c_T and c_E for the `then` and `else` branches respectively of a conditional statement c. For brevity, we sometimes refer to an execution path simply as a “path”.

Definition 1. *Let* `x` *be a variable,* `e` *an expression,* `c` *a Boolean expression, P a predicate, and P[*`e/x`*] the simultaneous substitution of* `x` *with* `e` *in P. The* weakest precondition `wp(`π`,`P`)` *of π with respect to P is defined inductively as follows:*

Assignment: $\mathtt{wp(x=e,}\ P\mathtt{)} = P\mathtt{[e/x]}$.
Conditional: $\mathtt{wp(if(c)}_T\mathtt{,}\ P\mathtt{)} = P \wedge \mathtt{c}$; $\mathtt{wp(if(c)}_E\mathtt{,}\ P\mathtt{)} = P \wedge \neg\mathtt{c}$.
Nondeterminism: $\mathtt{wp(if(*)}_T\mathtt{,}\ P\mathtt{)} = \mathtt{wp(if(*)}_E\mathtt{,}\ P\mathtt{)} = P$.
Sequence: $\mathtt{wp(}s_1;s_2\mathtt{,}\ P\mathtt{)} = \mathtt{wp(}s_1\mathtt{,\ wp(}s_2\mathtt{,}\ P\mathtt{))}$.

Given an execution path $\pi = \langle s_1, s_2, \ldots, s_n \rangle$, we use $\pi^i = s_i$ to denote the i-th statement of π, and $\pi^{i,j} = s_i, \ldots, s_j$ to denote the segment of π between i and j. Assume now that π^n, the last statement of π, is either c_T or c_E. If $\pi^n = c_T$, then it is impossible for any execution path with prefix $\pi^{1,n-1}$ to take the `else`-branch at π^n. That is, any execution path that has ρ as a prefix, where $\rho^i = \pi^i (1 \leq i < n)$ and $\rho^n \neq \pi^n$, is infeasible. Because of this, we say that ρ is an *infeasible sub-path.*

Let ρ be an infeasible sub-path of length n where ρ^n is a conditional c. We use $wp(\rho)$ to denote $wp(\rho^{1,n-1}, c)$, and $wp(\rho) = false$ as ρ is infeasible. According to Definition 1, assuming that ρ contains $t < n$ conditionals in addition to c, we have:

$$wp(\rho) = c' \wedge (c'_1 \wedge c'_2 \ldots \wedge c'_t) = false$$

where c' is ρ^n transformed through transitive variable substitutions, and similarly each c'_l is a transformed deterministic predicate in s_l: $(c_l)_{T/E}$ $(1 \leq l \leq t)$. More formally, given a formula F, we use F' to denote the formula in wp that is transformed from F. The definition is transitive in that both $F' = F(e/v)$ and $F'(e_2/v_2)$ are *transformed formulae* from F.

2.3 Learning From Infeasible Sub-paths

Upon encountering a new execution path, the `DPR-MC` algorithm collects information about infeasible sub-paths at deterministic predicates by using the weakest precondition computation presented in the previous section. We now analyze the reasons behind the infeasibility of such paths in order to provide useful information for pruning unexplored execution paths.

Since $wp(\rho)$ is unsatisfiable, there must exist an unsatisfiable subformula $wp_{us}(\rho)$ that consists of a subset of clauses $\{c', c'_1, c'_2 \ldots, c'_t\}$.

Definition 2. *A* minimally unsatisfiable subformula *of* $wp(\rho)$*, denoted by* $mus(\rho)$*, is a subformula of* $wp(\rho)$ *that becomes satisfiable whenever any of its clauses is removed. A* smallest cardinality MUS *of* $wp(\rho)$*, denoted by* $smus(\rho)$*, is an MUS such that for all* $mus(\rho)$*,* $|smus(\rho)| \leq |mus(\rho)|$*.*

In general, any unexplored paths that contain $mus(\rho)$ are infeasible and can be pruned. $wp(\rho)$ can have one or more MUSes; as a matter of succinctness, we keep track of $smus(\rho)$ for pruning purposes.

Next, we need to identify which statements are responsible for ρ's infeasibility and thus $smus(\rho)$.

Definition 3. *A* transforming statement *of a predicate c is an assignment statement* `s: v = e` *such that variable* v *appears in the transitive support of* c*.*

For example, the statement `s1:x = y+1` is a transforming statement of the condition $c : (x > 0)$, since $wp(s1, c)$ produces $c' : (y + 1 > 0)$. During weakest precondition computations, only assignment statements can transform an existing conjunct c into a new conjunct c'. Branching statements can only add new conjuncts to the existing formulae, but cannot transform them. Given an execution path $\pi^{i,j} = s_i, \ldots, s_j$, we use $ts(\pi^{i,j}, c) \subseteq \{s_i, \ldots, s_j\}$ to denote the transforming statements for c.

Definition 4. *We define the* explanation *of the infeasibility of ρ to be the set of transforming statements $explain(\rho) = \{s \mid s \in ts(\rho, smus(\rho))\}$.*

2.4 Pruning Unexplored Paths

In this section we use examples to illustrate how to prune the path search space based on information obtained after learning.

The SSA form of the program of Figure 1 is represented graphically to its right. Assume the first explored execution π (highlighted in the figure) takes the `then`-branches at the two nondeterministic `if` statements. We would like to learn from π to prune unexplored paths. In the example, $\pi = \langle x_1 = 5, y_1 = 8, z_1 = 20, *, y_2 = y_1 - 1, y_4 = y_2, x_2 = 2x_1, *, z_2 = z_1 - 1, z_4 = z_2, \neg(x_2 > 10), halt\rangle$, which implies the infeasible sub-path $\rho = \langle x_1 = 5, y_1 = 8, z_1 = 20, *, y_2 = y_1 - 1, y_4 = y_2, x_2 = 2x_1, *, z_2 = z_1 - 1, z_4 = z_2, x_2 > 10\rangle$. According to Definition 1, we have:

$$wp(\rho) = (x_1 = 5) \wedge (y_1 = 8) \wedge (z_1 = 20) \wedge (true) \wedge (true) \wedge (2x_1 > 10) = false$$

The first three conjuncts come from the initial variable assignments and the next two ($true$) come from the nondeterministic conditionals. The last conjunct $2x_1 > 10$ is due to the deterministic conditional $x_2 > 10$ and the assignment $x_2 = 2x_1$. A decision procedure can decide $smus(\rho) = (2x_1 > 10) \wedge (x_1 = 5)$. The explanation for ρ's infeasibility is $explain(\rho) = \{x_1 = 5, x_2 = 2 * x_1\}$. Therefore, we learned that any path containing these two assignments will not satisfy $x_2 > 10$; that is, any execution that contains $explain(\rho)$ can only take the `else`-branch to the conditional $x_2 > 10$. Since all the four possible paths contain $explain(\rho)$, none can reach the `abort` statement, which requires a path through the `then`-branch of the conditional $x_2 > 10$. Therefore, with DPR, a proof is obtained after only one execution.

A question that naturally arises from the example is what happens if a variable assigned in $explain(\rho)$ is subsequently reassigned? The answer is that if a variable is reassigned at s_i, then s_i will be included in $explain(\rho)$ if it is considered part of the explanation to ρ's infeasibility. For example, consider the program `foo2` which is the same as program `foo` of Figure 1 except for an additional assignment $x = x + 1$. The SSA form of `foo2` and its graphical representation is shown in Figure 2. Due to the new assignment $x = x + 1$ on Line 11, we need to add $x_4 = \varphi(x_2, x_3)$ on Line 12 to decide which version of x to use on Line 14. Assume the first execution, as highlighted in Figure 2, is $\pi_2 = \langle 0 : x_1 = 5, y_1 = 8, z_1 = 20, 1_T : *, 2 : y_2 = y_1 - 1, 5 : y_4 = y_2, 6 : x_2 = 2x_1, 7_T : *, 8 : z_2 = z_1 - 1, 12 : x_4 = x_2, 13 : z_4 = z_2, 14_E : \neg(x_4 > 10), 15 : halt\rangle$. From this, we can infer the infeasible execution segment $\rho_2 = \langle 0 : x_1 = 5, y_1 = 8, z_1 = 20, 1_T : *, 2 : y_2 = y_1 - 1, 5 : y_4 = y_2, 6 : x_2 = 2x_1, 7_T : *, 8 : z_2 = z_1 - 1, 12 : x_4 = x_2, 13 : z_4 = z_2, 14_T : x_4 > 10\rangle$. Based on an analysis similar to that used in the previous example, we have:

$$wp(\rho_2) = ((x_1 = 5) \wedge (y_1 = 8) \wedge (z_1 = 20) \wedge (true) \wedge (true) \wedge (2x_1 > 10)) = false$$

```
foo2_SSA( ){
    x1=5,y1=8,z1=20;
    if (*)
        y2 = y1-1;
    else
        y3 = y1+1;
    y4 = φ(y2,y3);
    x2 = 2*x1;
    if (*)
        z2 = z1-1;
    else
        z3 = z1+1;
        x3= x2+1;
    x4 = φ(x2,x3);
    z4 = φ(z2,z3);
    if (x4 >10)
        abort;
    else
        halt;
}
```

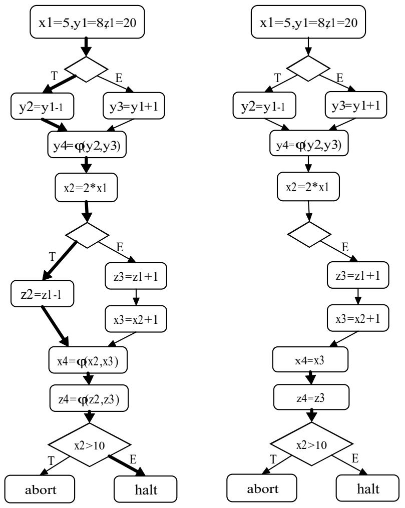


Fig. 2. A C program in SSA form (left), its graphical representation with a highlighted execution path (middle), and the remaining paths after learning from the highlighted path

Although it results in the same $smus(\rho_2) = (2x_1 > 10) \wedge (x_1 = 5)$ as $smus(\rho_1)$, the explanation to $smus(\rho_2)$ is different: $explain(\rho_2) = \{x_1 = 5, x_2 = 2x_1, x_4 = x_2\}$. As a result, we can prune fewer paths than in the previous example of Figure 1. Figure 2(right) shows the remaining paths after pruning. Both of the remaining paths take the `else`-branch at the second nondeterministic `if` statement, which will go through $x_4 = x_3$. They cannot be pruned because neither path contains the statement $x_4 = x_2$ of $explain(\rho_2)$.

2.5 Path Reduction Algorithm

Algorithm 2 gives the pseudo-code that our `DPR-MC` algorithm uses in order to drive the execution of the program under verification along the path determined by a given k-oracle R. If the current statement s_i is an `abort` statement (Lines 3-4), an execution with a bug is found and the algorithm terminates. If s_i is a `halt` statement (Lines 5-6), the current execution is completed. An assignment is performed if s_i is an assignment statement (Lines 7-8). If s_i is a nondeterministic conditional (Lines 9-12), the algorithm checks if the threshold k has already been reached. If not, the algorithm follows the branch specified by oracle $R[j]$ and increase the value of j by 1; otherwise the algorithm breaks from the loop. If s_i is a deterministic conditional c (Lines 13-17), the value of c is computed and the corresponding branch is taken. Meanwhile, DPR is performed on the branch not taken, as shown in Algorithm 3, if the taken branch cannot reach the `abort` statement. Finally, the completed execution is removed from the unexplored oracle set (Line 20).

Algorithm 2. ExecuteFollowOracle(k-Oracle R, Set $\mathcal{R}$, int k)

```
1: $i = j = 0$;
2: **while** true **do**
3:    **if** $s_i$== abort **then**
4:       exit(“report bug trace $\langle s_1, \ldots, s_i \rangle$”);
5:    **else if** $s_i$ == halt **then**
6:       break;
7:    **else if** $s_i$ is an assignment **then**
8:       Perform the assignment;
9:    **else if** $s_i$ is a nondeterministic conditional **then**
10:      if $j == k$ break;
11:      follow oracle $R[j]$;
12:      $j++$;
13:   **else if** $s_i$ is deterministic conditional $c$ with value *true* **then**
14:      LearnToPrune($\langle s_1, \ldots, s_{i-1}, \neg c \rangle$, $\mathcal{R}$) if then-branch cannot reach abort;
15:   **else if** $s_i$ is deterministic conditional $c$ with value *false* **then**
16:      LearnToPrune($\langle s_1, \ldots, s_{i-1}, c \rangle$, $\mathcal{R}$) if else-branch cannot reach abort;
17:   **end if**
18:   $i++$;
19: **end while**
20: $\mathcal{R} = \mathcal{R} - \{R\}$;
```

The SMT-based learning procedure is given in Algorithm 3. The meaning of, and reason for, each statement, i.e., weakest-precondition computation, SMUS and transforming statements, have been explained in previous sections.

Algorithm 3. LearnToPrune(InfeasibleSubPath ρ, Set $\mathcal{R}$)

```
1: $w = wp(\rho)$; // Perform weakest precondition computation
2: $s = smus(w)$ // Compute smallest cardinality MUS
3: $e = explain(s)$; // Obtain transforming statements
4: $\mathcal{R} = prune(\mathcal{R}, e)$; //Remove all oracles in $\mathcal{R}$ that define paths containing $e$
```

3 Implicit Oracle Enumeration Using SAT

One problem with Algorithm 1 is the need to save in $\mathcal{R}$ all k-oracles when model checking commences, the number of which can be exponential in k. In order to avoid this complexity, we show how Boolean formulae can be used to symbolically represent k-oracles.

Our discussion of the symbolic representation of k-oracles will be centered around loop-unrolled control flow graphs (CFGs), which can be viewed as directed acyclic graphs whose nodes are program statements and whose edges represent the control flow among statements. We shall assume that every loop-unrolled CFG has a distinguished root node. The *statement depth* of a loop-unrolled CFG is the maximum number of statements along any complete path

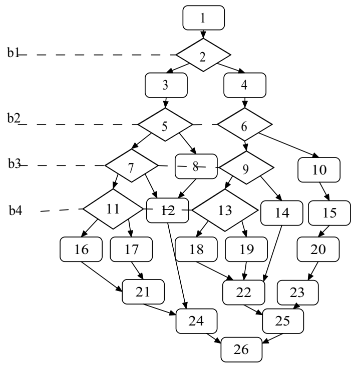


Fig. 3. An example control flow graph

from the root. The *oracle depth* of a loop-unrolled CFG is the maximum number of nondeterministic conditional nodes along any complete path from the root.

Figure 3 depicts a typical loop-unrolled CFG, where each node in the CFG has a unique index. Diamond-shaped nodes correspond to nondeterministic conditionals and rectangles are used for other statement types. The statement depth of this CFG is 10. As for its oracle depth, there are 7 nondeterministic conditionals divided into 4 levels (indicated by dotted lines); i.e., its oracle depth is 4.

To encode the choice made along a particular execution path at each level, we introduce the Boolean variables b_1, b_2, b_3 and b_4, with positive literal b_i indicating the `then`-branch and negative literal $\neg b_i$ indicating the `else`-branch. For example, path $\langle 1, 2, 4, 6, 9, 13, 19, 22, 25, 26\rangle$ is captured by $\neg b_1 \wedge b_2 \wedge b_3 \wedge \neg b_4$.

In general, a loop-unrolled CFG will have k levels of nondeterministic conditionals, and we will use k-oracles to explore its path space, with each k-oracle represented as a bit vector of the form $R = \langle b_1, b_2, \ldots, b_k\rangle$. As such, the valuation of Boolean variable b_i indicates an oracle's choice along an execution path at level i, and we call b_i an *oracle choice variable* (OCV). Such considerations lead to a symbolic implementation of the oracle space in which we use Boolean formulae over $b_i (1 \leq i \leq k)$ to encode k-oracles. For example, the Boolean formula $b_1 b_2 b_3 b_4 + \neg b_1 b_2 \neg b_3$ encodes two paths through the CFG of Figure 3: $\langle 1, 2, 3, 5, 7, 11, 16, 21, 24, 26\rangle$ and $\langle 1, 2, 4, 6, 9, 14, 22, 25, 26\rangle$. In order to use modern SAT solvers, we maintain such Boolean formulae in conjunctive normal form (CNF).

Algorithm 4 presents a SAT-based implementation of Algorithm 1. It maintains a CNF $\mathcal{B}$ over k OCVs $\{b_1, b_2, \ldots, b_k\}$. Initially, $\mathcal{B}$ is a tautology; the while-loop continues until $\mathcal{B}$ becomes unsatisfiable. Inside the while-loop, we first use a SAT solver to find a k-oracle that is a solution of $\mathcal{B}$, and then perform the program execution determined by the oracle. Algorithm 4 is essentially the same as Algorithm 1 except that: 1) oracle R and set $\mathcal{R}$ are represented symbolically by $\widehat{b}$ and $\mathcal{B}$, respectively; and 2) function calls to `LearnToPrune` (in algorithm `ExecuteFollowOracle`) are replaced by function calls to `SATLearnToPrune`, whose pseudo-code is given in Algorithm 5.

Algorithm 4. DPR-SATMC(PROGRAM C, INT k)

```
1: Let $b_i (1 \leq i \leq k)$ be $k$ OCV variables, where $k$ is $C$'s oracle depth;
2: CNF $\mathcal{B} = true$;
3: while $\mathcal{B}$ is satisfiable do
4:    Obtain a $k$-oracle $\widehat{b} = \langle \widehat{b}_1 \widehat{b}_2 \dots \widehat{b}_k \rangle$ which is a solution of $\mathcal{B}$;
5:    ExecuteFollowOracle($\widehat{b}, \mathcal{B}, k$);
6: end while
7: exit("No bug found up to oracle-depth $k$");
```

Let s be an assignment statement in an infeasible sub-path ρ. We define OCV_s to be the conjunction of those signed (positive or negative) OCVs within whose scope s falls. Also, given ρ's set of transforming statements $explain(\rho) = \{s_1, \dots, s_t\}$, $OCV(explain(\rho)) = \wedge_{i=1}^{t} OCV_{s_i}$. Note that $OCV(\rho) \neq false$ as all statements in $explain(\rho)$ are along a single path. Further note that $explain(\rho)$ and $OCV(explain(\rho))$ can be simultaneously computed with one traversal of ρ: if a transforming statement s in $explain(\rho)$ is within the scope of a nondeterministic conditional, then the conditional's associated OCV variable is in OCV_s.

To illustrate these concepts, assume $explain(\rho) = \{1, 4, 22\}$ in the loop-unrolled CFG of Figure 3. Since node 1 can be reached from root node without going through any conditional branches, $OCV_1 = true$. Node 4 on the other hand is within the scope of the `else`-branch of nondeterministic conditional node 2 and thus $OCV_4 = \neg b_1$. Similarly, $OCV_{22} = \neg b_1 b_2$. Notice that the scopes of b_3 and b_4 close prior to node 22 and are therefore not included in OCV_{22}. Finally, $OCV(\rho) = OCV_1 \wedge OCV_4 \wedge OCV_{22} = \neg b_1 b_2$.

Algorithm 5 is our SAT-based implementation of Algorithm 3. $OCV(e)$ determines the set of paths containing all statements in $explain(\rho)$, and thus all paths that can be pruned. Let $OCV(e) = l_1 \wedge l_2 \wedge \dots \wedge l_m$, where l_i is a literal denoting b_i or $\neg b_i$. Adding $\neg OCV(e) = \neg l_1 \vee \neg l_2 \vee \dots \vee \neg l_m$ to the CNF formula $\mathcal{B}$ will prevent the SAT solver from returning any solution (k-oracle) that has been pruned. We refer to $\neg OCV(e)$ as a *conflict clause*.

Algorithm 5. SATLEARNTOPRUNE(INFEASIBLESUBPATH ρ, CNF $\mathcal{B}$)

```
1: $w = wp(\rho)$; // Perform weakest precondition computation
2: $s = smus(w)$; // Compute smallest cardinality MUS
3: $e = explain(s)$; // Obtain transforming statements
4: $b = OCV(e)$; //Obtain OCV on which $e$ depends
5: let $b = l_1 \wedge l_2 \wedge \dots \wedge l_m$ where $l_i$ is a literal for $b_i$ or $\neg b_i$;
6: $\mathcal{B} = \mathcal{B} \wedge (\neg l_1 \vee \neg l_2 \vee \dots \vee \neg l_m)$;
```

Note that the added conflict clause may prune multiple oracles, including the one just executed. Further note that when exploring a path by virtue of a given k-oracle, not all OCVs may be executed. For example, if the k-oracle in question is $b_1 \neg b_2 b_3 b_4$ in Figure 3, the actual execution path terminates after $\neg b_2$. In this case, the added conflict clause is $(\neg b_1 \vee b_2)$ instead of $(\neg b_1 \vee b_2 \vee \neg b_3 \vee \neg b_4)$.

To further illustrate Algorithms 4 and 5, consider once again the program of Figure 1. Suppose that the first path π_1 to be explored is the highlighted one in the figure. In this case, the infeasible sub-path ρ_1 to be considered is the same as π_1 except that the then-branch of the final deterministic conditional is taken leading to the `abort` statement. We then have that $smus(\rho_1) = (2x_1 > 10) \wedge (x_1 = 5)$ and the explanation for ρ_1's infeasibility is $explain(\rho_1) = \{x_1 = 5, x_2 = 2 * x_1\}$. Moreover, $OCV(e_1) = true$ as neither of the statements in $e_1 = explain(\rho)$ are in the scope of a nondeterministic conditional. The resulting conflict clause is *false* and adding (conjoining) it to $\mathcal{B}$ renders $\mathcal{B}$ unsatisfiable; i.e., all remaining paths can be pruned.

Consider next the program of Figure 2 and its highlighted execution π_2. As explained in Section 2, $smus(\rho_2) = smus(\rho_1)$, where ρ_2 is the infeasible sub-path corresponding to π_2. However, the explanation for $smus(\rho_2)$, $explain(\rho_2) = \{x_1 = 5, x_2 = 2x_1, x_4 = x_2\}$, is different. Furthermore, $OCV(e_2) = b_2$, where $e_2 = explain(\rho_2)$, since the assignment $x_4 = x_2$ is within the scope of the `then`-branch of the second nondeterministic conditional. We thus add conflict clause $\neg b_2$ to $\mathcal{B}$, which results in the two remaining paths after pruning illustrated in Figure 2(right), both of which take the `else`-branch at the second nondeterministic conditional.

Theorem 1. *(Soundness and Completeness). Let C be a CFG that is loop-unrolled to statement depth D, and let ϕ be a safety property, the violation of which is represented by an* `abort` *statement in C. Then algorithm* `DPR-MC` *reports that the* `abort` *statement is reachable if and only if C violates ϕ within statement depth D.*

Proof.

- Soundness*: Algorithm* `DPR-MC` *reports a counterexample only when a statement in C that violates ϕ is reached within depth D. Since procedure* `ExecuteFollowOracle`*, which is used to follow the path of the counterexample, is precise, the reported counterexample is an acutal counterexample in C.*
- Completeness*: Algorithm* `DPR-MC` *starts from a set $\mathcal{P}$ containing all paths in C up to depth D. A path is removed from $\mathcal{P}$ only if it has been executed or shown by the learning procedure not to violate φ. The fact that* `DPR-MC` *terminates without counterexamples only if $\mathcal{P}$ becomes empty proves that if $C \not\models \phi$ within depth D, a counterexample will be reported. In the symbolic implementation, the completeness of* `DPR-MC` *also relies on the completeness of SAT solvers. That is, the SAT solver declares unsatisfiability only if the CNF instance has no solutions.*

4 Experimental Evaluation

In order to assess the effectiveness of the DPR technique in the context of bounded model checking, we conducted several case studies involving well-known randomized algorithms. All results were obtained on a PC with a 3 GHz Intel

Table 1. Bounded model checking with DPR of Randomized MAX-3SAT

vars	clauses	paths	explored	pruned	time w DPR(s)	time w/o DPR(s)
9	349	512	44	468	5.44	3.86
10	488	1024	264	760	13.77	7.61
11	660	2048	140	1908	9.67	15.58
12	867	4096	261	3835	14.53	30.59
13	1114	8192	1038	7154	49.61	70.10
14	1404	16384	965	15419	54.05	150.32
15	1740	32768	337	32431	25.58	300.80
16	2125	65536	2369	63167	49.32	Timeout
17	2564	131072	2024	129048	184.91	Timeout
18	3060	262144	1344	260800	175.34	Timeout
19	3615	524288	669	523619	110.14	Timeout

Duo-Core processor with 4 GB of RAM running Fedora Core 7. We set a time limit of 500 seconds for each program execution.

In the first case study, we implemented a randomized algorithm for the MAX-3SAT problem. Given a 3-CNF formula (i.e., with at most 3 variables per clause), MAX-3SAT finds an assignment that satisfies the largest number of clauses. Obtaining an exact solution to MAX-3SAT is NP-hard. A randomized approximation algorithm independently sets each variable to 1 with probability 0.5 and to 0 with probability 0.5, and the number of satisfied clauses is then determined. In our implementation, we inserted an unreachable `abort` statement; as such, all paths have to be explored to prove the absence of any reachable `abort` statement. Table 1 presents our experimental results for the randomized MAX-3SAT algorithm. Each row of the table contains the data for a randomly generated CNF instance, with Columns 1 and 2 listing the number of variables and clauses in the instance, respectively. Columns 3-5 respectively show the total number of execution paths, the number explored paths, and the number of pruned paths, with the sum of the latter two equal to the former. Finally, Columns 6-7 present the run time with DPR and the run time of executing all paths without DPR. From these results, we can observe that DPR is able to prune a significant number of the possible execution paths. Furthermore, the larger the CNF instance, the more effective dynamic path reduction is.

Table 2. Bounded model checking with DPR of NFA for floating-point expressions

Benchmark			With DPR			Without DPR	
length	valid	paths	explored	pruned	time(s)	explored	time(s)
13	yes	8192	22	8166	0.707	2741	0.085
14	yes	16384	28	16356	0.845	10963	0.144
18	yes	262144	39	262105	2.312	175403	7.285
20	yes	1048576	29	1048542	4.183	350806	6.699
21	yes	2097152	26	2097097	4.202	175403	4.339
11	no	2048	15	2033	1.69	2048	10.027
13	no	4096	13	4083	0.52	4096	16.607
14	no	16384	8	16376	0.84	16384	53.358
20	no	1048576	28	1048548	3.32	-	Timeout

In our second case study, we implemented an algorithm that uses a Nondeterministic Finite Automaton (NFA) to recognize regular expressions for floating-point values of the form $[+]?[0-9]+\backslash.[0-9]+$. We encoded the accept state as an `abort` statement and verified whether it is reachable. Table 2 contains our experimental results on nine input sentences, among which five are valid floating-point expressions and four are not. Columns 1 and 2 give the length of the input and whether or not it is accepted by the NFA. Column 3 lists the total number of execution paths Columns 4-6 contain the results using DPR, i.e. the number explored paths, the number of pruned paths and the run time. Columns 7 and 8 list the number of explored paths and run time without DPR. Note that in the case of a valid floating-point expression, the number of explored paths without DPR may not be the same as the number of total paths since the accept state is reached before exploring the remaining paths. As in the MAX-3SAT case study, we can again observe a very high percentage of pruned paths, a percentage that grows with the instance size.

5 Related Work

With dynamic path reduction, we perform symbolic analysis on program executions in order to learn and subsequently prune infeasible executions. Concolic testing and related approaches [7,2,9] also uses symbolic analysis of program executions—in conjunction with random testing—to generate new test inputs that can systematically direct program execution along alternative program paths. While these approaches can handle nondeterminism by introducing a Boolean input variable for each nondeterministic choice, they do not attempt to learn and prune infeasible paths. In fact, these testing procedures generate all possible paths and, for each such path, pass to a constraint solver the relevant constraints to determine the path's feasibility. Consequently, DPR can be beneficially used to reduce the path space these procedures explore.

The optimizations realized by DPR are similar to those achieved by performing bounded model checking (BMC) with a SAT/SMT solver [8]. There are, however, significant differences between the two approaches. First, DPR is based on dynamic path execution whereas BMC is a form of abstraction-based software model checking. Consequently, BMC, unlike DPR, may miss subtle implementation errors and may not able to provide partial results. Second, the manner in which DPR and BMC utilize constraint solvers is different. BMC converts the complete program model (up to a predefined bound) to a SAT formula and relies on a constraint solver to check its satisfiability; DPR uses a constraint solver to analyze a single infeasible path. The memory consumption in DPR is therefore much less demanding than in BMC. Finally, BMC relies on nonchronological backtracking (NCB) [12], a feature of modern SAT/SMT solvers, to achieve pruning. NCB avoids redundant search by flipping the most recent open decision contributing to a conflict. Therefore, the learning that occurs in BMC takes place post-conflict. On the other hand, DPR starts from a particular path that ends with condition c, and learns, before any conflict happens, from the fact that the path that ends with not c is not taken. Therefore, DPR *pre-computes* the paths to explore, which is made possible by the knowledge of control and data flow information at the programming language level.

Similar to fault localization [14], DPR uses weakest-precondition and minimally-unsatisfiable-core computations to identity interesting (transforming) statements along an execution path. Program slicing also attempts to identify interesting program statements. Due to the imprecision of static dependence graphs, dynamic slicing algorithms [17] have been proposed to build more accurate dynamic dependence graphs, which lead to more precise results. However, even though the dependence graphs capture the dependence relation among statements, it contains no information about values and therefore cannot offer precise answer to questions involving values. For example, given an execution path $\pi = \langle x = 10; y = x; w = 0; w = w * y; assert(w! = 0); \rangle$, dynamic program slicing will determine that all the statements are responsible for the assertion failure. On the other hand, DPR will only pick $w = 0$ and $w = w * y$ as the reason for the assertion failure.

6 Conclusions

We have presented the new technique of dynamic path reduction (DPR) for software model checking. SMT-based learning techniques allow DPR to prune infeasible execution paths while model checking sequential software systems with nondeterministic conditionals. Our approach uses weakest-precondition and minimally-unsatisfiable-core computations to reveal the interesting (transforming) statements behind infeasible sub-paths. By determining the oracle control variables associated with these statements, unexplored paths containing the same unsatisfiable core can be efficiently and dynamically pruned. Our preliminary experimental results show that DPR can prune a significant percentage of execution paths, a percentage that grows with the instance size.

The language we currently handle is a subset of C allowing only one procedure and assignments, loops and (and possibly nondeterministic) conditional statements. There are no constraints placed on conditionals, but the constraint solver is able to handle linear constraints only. While we can analyze certain applications, future work will seek to extend the DPR technique to more general programs, including those with input statements.

References

1. Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: SIGPLAN Conference on Programming Language Design and Implementation, pp. 203–213 (2001)
2. Cadar, C., Ganesh, V., Pawlowski, P., Dill, D., Engler, D.: EXE: automatically generating inputs of death. In: ACM conference on Computer and communications security (CCS) (2006)
3. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)

4. Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
5. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032. Springer, Heidelberg (1996)
6. Godefroid, P.: Model checking for programming languages using VeriSoft. In: POPL 1997: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 174–186. ACM Press, New York (1997)
7. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI) (2005)
8. Ivancic, F., Yang, Z., Ganai, M., Gupta, A., Ashar, P.: Efficient sat-based bounded model checking for software verification. Theoretical Computer Science 404(3) (2008)
9. Majumdar, R., Sen, K.: Hybrid concolic testing. In: International Conference on Software Engineering (ICSE) (2007)
10. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers, Boston (1994)
11. Peled, D.: All from one, one for all: on model checking using representatives. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 409–423. Springer, Heidelberg (1993)
12. Silva, J.P.M., Sakallah, K.A.: GRASP—a new search algorithm for satisfiability. In: ICCAD 1996: Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design, pp. 220–227 (1996)
13. Valmari, A.: Stubborn sets for reduced state generation. In: Rozenberg, G. (ed.) APN 1990. LNCS, vol. 483, pp. 491–515. Springer, Heidelberg (1991)
14. Wang, C., Yang, Z.-J., Ivančić, F., Gupta, A.: Whodunit? Causal analysis for counterexamples. In: Graf, S., Zhang, W. (eds.) ATVA 2006. LNCS, vol. 4218, pp. 82–95. Springer, Heidelberg (2006)
15. Wang, C., Yang, Z., Ivancic, F., Gupta, A.: Disjunctive image computation for software verfication. ACM Transactions on Design Automation of Electronic Systems 12(2) (2007)
16. Yang, Z., Wang, C., Gupta, A., Ivancic, F.: Model checking sequential software programs via mixed symbolic analysis. In: ACM Transactions on Design Automation of Electronic Systems (TODAES) (to appear)
17. Zhang, X., Gupta, R., Zhang, Y.: Precise dynamic slicing algorithms. In: IEEE/ACM International Conference on Software Engineering, pp. 319–329 (2003)

Automatic Generation of Error Messages for the Symbolic Execution of EB3 Process Expressions

Jérémy Milhau, Benoît Fraikin, and Marc Frappier

Université de Sherbrooke, Département d'Informatique
2500 Boulevard Université, Sherbrooke, Québec, Canada, J1K 2R1
{Jeremy.Milhau,Benoit.Fraikin,Marc.Frappier}@USherbrooke.ca

Abstract. This paper describes an algorithm to automatically generate error messages for events refused by a process expression. It can be used in the context of an information system specified with the EB^3 method. In this method, a process expression is used to describe the valid traces of events that the information system must accept. If a user submits an event which is rejected by this process expression, our algorithm produces an error message explaining why the event has been rejected; it also suggests which event should be submitted in order to correct the error.

1 Introduction

Information systems (IS) are commonly used nowadays in organizations, but their development is expensive and requires a long-term vision for their integration. With the aim of reducing cost and time needed to develop such systems, the Automated Production of Information Systems (APIS) project [1] was launched to provide an efficient way to generate ISs from formal specifications. As part of the APIS project, the Entity-Based Black-Box method (EB^3) [2,3] provides mechanisms to write formal specifications that describe the input-output behavior of the IS. A specification includes a process expression (PE) that describes the valid traces of input events that the IS must accept.

Fraikin and Frappier have shown that a PE can be efficiently executed by the EB^3 Process Algebra Interpreter (EB^{3}PAI) [4], the core of the generated IS. If, for one reason or another, the user input, which is called the query henceforth, does not comply with the specification, EB^{3}PAI rejects it and preserves the state of the IS.

In such a case, an error message [5] must be issued to report to the user that his query is not valid, and explain why and how he can modify his query to comply with the IS specification. In a traditional implementation of an IS, error messages are determined and implemented by a programmer. Since EB^{3}PAI uses symbolic execution based on the operational semantics of the EB^3 process algebra, the state of the system is represented by the abstract syntax tree (AST) of the PE. Hence, we can not manually determine the error message to produce when an event is rejected. Consequently, we have derived an algorithm that produces an error message through an analysis of the AST of the PE. This problem is somewhat similar to the generation of error messages within a compiler (*e.g.* a Java compiler). The PE of the IS specification corresponds to the programming language addressed by the compiler (*e.g.*, Java for a Java compiler). The user inputs correspond to the program text (*i.e.* the Java program submitted to the Java compiler). An invalid event corresponds to a syntax error in the program text. However, there are major differences: a compiler is built for a specific programming language (*e.g.* Java); EB^{3}PAI must be able to execute any PE. Since a compiler is built for a single language, the compiler designer can exploit the syntax of the language to determine the error message. Error message generation in a compiler is an art that the compiler designer must master [6]. In our case, we must devise a generic algorithm, which works for any PE.

2 Background

The EB^3 process algebra is similar to several other process algebra like CSP [7], CCS, ACP and Lotos. We shall briefly introduce it here; the reader is referred to [2] for a complete description. An elementary PE is either an internal action λ, which plays the same role as ϵ in regular expressions, or an action $\mathsf{a}(\,t_1, \ldots, t_n\,)$, where a is an action label and t_i denotes a constant or a variable. Compound PEs are built using the following operators: $E_1 \,\textbf{.}\, E_2$ (sequence, also called concatenation), $E_1 \mid E_2$ (choice), $E_1{}^*$ (Kleene closure of E_1), $E_1 |[\Delta]| \, E_2$ (synchronization over the set Δ) and $p \Longrightarrow E$ (guard p over the PE E). A quantified version of the choice $|$ and the synchronization $|[\Delta]|$ can also be used. Other operators are built from these operators: $E_1 \; ||| \; E_2$ (interleaving operator) which is defined as $E_1 |[\emptyset]| \, E_2$, and $E_1 \; || \; E_2$ (parallel operator) defined as $E_1 |[\Delta]| \, E_2$ where Δ is the intersection of the alphabets of the operands. The PE ΓE denotes the application of environment Γ to E; Γ is a substitution $[\![x_1 := y_1, \ldots x_n := y_n]\!]$ which denotes the values of variables x_i in E.

2.1 Symbolic Execution of EB^3 Process Expression

EB^{3}PAI uses symbolic execution based on the operational semantics of the EB^3 process algebra. For instance, the PE $\mathsf{a}\,\textbf{.}\,(\mathsf{b} \mid \mathsf{c})\,\textbf{.}\,\mathsf{d}$ can execute a and transform into PE $(\mathsf{b} \mid \mathsf{c})\,\textbf{.}\,\mathsf{d}$, which we note $\mathsf{a}\,\textbf{.}\,(\mathsf{b} \mid \mathsf{c})\,\textbf{.}\,\mathsf{d} \xrightarrow{\mathsf{a}} (\mathsf{b} \mid \mathsf{c})\,\textbf{.}\,\mathsf{d}$; this is called a transition and it is computed by EB^{3}PAI from the inference rules of the operational semantics. PE are represented by their AST in EB^{3}PAI.

2.2 A Specification

A simple example of IS specification is the library. The library can acquire and discard books, and members can join or leave membership of the library. If a member wants to borrow a book, he must first reserve it if the book is already borrowed. A member can then renew its loans and return a borrowed book. The first person on the reservation list can borrow the book when the previous borrower returned it and anyone can cancel its reservation at anytime. Some statistics such as the current borrower of a book, the number of loans for a member and a list of all borrowers by category can be generated.

The specification of Fig. 1 describes the behavior of an IS which aims to monitor a library for both members and books. This specification follows commonly used patterns [2] for entities (book and member) and relationships (loan and reservation).

This specification describes the life cycle of members and books that are then put in parallel execution. The process **member** describes a member whose key is mId.

$$\begin{aligned}
\mathbf{main} = \ & (\ |||\ bId \in \text{BOOKID} \ : \ \mathbf{book}(bId)^*\ ) \\
& ||\ (\ |||\ mId \in \text{MEMBERID} \ : \ \mathbf{member}(mId)^*\ ) \\
& ||\ \mathsf{DisplayBorrowerByCategory}(\ )^*
\end{aligned}$$

$$\begin{aligned}
\mathbf{book}(bId : \text{BOOKID}) = \ & \mathsf{Acquire}(bId, \_) \\
& .\ (\quad (|\ mId \in \text{MEMBERID} \ : \ \mathbf{loan}(mId, bId))^* \\
& \quad ||\ (\ |||\ mId \in \text{MEMBERID} \ : \ \mathbf{reservation}(mId, bId)^*) \\
& \quad ||\ \mathsf{DisplayCurrentBorrower}(bId)^* \\
& \ ) \\
& .\ \mathsf{Discard}(bId)
\end{aligned}$$

$$\begin{aligned}
\mathbf{member}(mId : \text{MEMBERID}) = \ & \mathsf{Join}(mId) \\
& .\ (\quad (\ |||\ bId \in \text{BOOKID} \ : \ \mathbf{loan}(mId, bId)^*\ ) \\
& \quad ||\ (\ |||\ bId \in \text{BOOKID} \ : \ \mathbf{reservation}(mId, bId)^*) \\
& \quad ||\ \mathsf{DisplayNumberOfLoans}(mId)^* \\
& \ ) \\
& .\ \mathsf{Leave}(mId)
\end{aligned}$$

$$\begin{aligned}
& \mathbf{loan}(mId : \text{MEMBERID}, bId : \text{BOOKID}) = \\
& \quad (\ \mathsf{Lend}(mId, bId)\ |\ \mathsf{Take}(mId, bId)\ )\ .\ \mathsf{Renew}(mId, bId)^*\ .\ \mathsf{Return}(mId, bId)
\end{aligned}$$

$$\begin{aligned}
& \mathbf{reservation}(mId : \text{MEMBERID}, bId : \text{BOOKID}) = \\
& \quad \mathsf{Reserve}(mId, bId)\ .\ (\ isFirst(trace, mId, bId) \implies \mathsf{Take}(mId, bId) \\
& \qquad\qquad\qquad\qquad\qquad |\ \mathsf{Cancel}(mId, bId)\ )
\end{aligned}$$

Fig. 1. EB^3 PE of a library

By using $|||\ mId \in \text{MEMBERID} \ : \ \mathbf{member}(mId)^*$, the specification allows multiple **member** processes with different mId keys to run simultaneously. The same technique is used for the **book** process that allows multiple instances of **book** with different bId to run simultaneously.

Both sets of processes run in parallel execution and synchronize over the shared actions of the **loan** and the **reservation** processes. Since one book can not be borrowed by more than one member at once, the quantification for **loan** in the **book** process is over a choice.

2.3 Error Types

Two categories of errors can be detected: syntax errors and execution errors.

Syntax Errors. Syntax errors are the simplest to detect and manage. They correspond to a violation of the signature of actions defined in the EB^3 specification. For instance, they include query parameters with improper types, an invalid number of parameters (*i.e.* missing or extra parameters), and an invalid action label. The production of error messages for these cases is trivial and not described in this paper.

Execution Errors. Execution errors denote syntactically correct queries which can not be accepted by the current PE. This paper focuses on the generation of error messages for this kind of errors.

3 Identifying the Cause of an Execution Error

This section describes the algorithm used to identify the cause of an error. Note that the PE is supposed to be correct; it is the user query which is incorrect. However, we will speak about the "causes" of an error in the PE, in order to identify why the query is rejected. The cause of an error is essentially one or several operators in the PE which can not accept the user query.

For the sake of illustration[1], the terminology of police investigation is used to describe the process of finding of the cause of an execution error. The causes of an error are referred as *guilty* operators that are designated among *suspect* operators. Some operators may also have an *alibi* that removes them from the suspect list. There are two classes of alibi: acquired and inherent. An acquired alibi depends upon the context of execution. It is called an *environmental alibi*. An inherent alibi is always valid whatever the crime is, *i.e.* whatever the context is. Operators with inherent alibis are referred below as *permissive operators*. The operators that belong to this class will be safe from the investigation due to their semantics that prove they can not be designated as guilty.

The next algorithm, called $main$, computes information from the current state of the IS and the query from the user to deduce a list of guilty operators. It describes the needed steps of the investigation that lead to a *verdict*, *i.e.* a complete error message. The functions called by $main$ are described in the rest of Sect. 3.

Algorithm 1. $main(E, \sigma)$

Description: *The main algorithm for the generation and display of error messages.*
input E**:** *An* EB^3 *PE that describes the current state of an IS.*
input σ**:** *A query from the user refused by* E*.*

```
$main(E, \sigma)$
begin
    let $V$ be the list of victims, $v = Victims(E, \sigma)$,
    let $V'$ be the list of victims with no environmental alibi,
        $V' = filter(V, \lambda x.noEnvironmentalAlibi(x, E, \sigma))$,
    let $W$ be the list of list of witnesses, $W = map(\lambda x.Witnesses(x, E, \sigma), V')$,
    let $S$ be the list of list of suspects, $S = map(\lambda x.Suspects(x), W)$
    let $C$ be the list of culprits, $C = map(\lambda x.Culprit(x), S)$
    foreach $c$ in $C$ do
        $display(getVerdictFrom(c, \sigma, E))$
    done
end
```

where map *and* $filter$ *are the classical list operators in functional programming and* $getVerdictFrom$ *creates the error message; its algorithm is described in Sect. 4.*

[1] And also for the fun of it.

The reader is invited to consult [8] for complete details about the implementation of this algorithm in Caml.

3.1 Victims

Considering the query entered by the user of the IS, a list of all the potential victims of the error is made. these are the actions of the PE that share with the query the same action label. The intuition is that if a PE can execute the query, there must exist a leaf in the AST that matches this query. If there is no leaf, that matches this query, then it will never be possible to execute the query. When such a leaf exists, we can start analysing why it can be executed. From this list are removed actions that do not comply with the restrictions imposed by the parameters of the query. Indeed, the parameters of the query may have values that are not compatible with the values of corresponding parameters in the PE. That is why these actions with environmental alibis are removed from the list of victims.

Definition 1. ***Environmental alibi of an action relative to a query***
Let $\sigma = \mathsf{a}(\,v_1, ..., v_n\,)$ be a query. An action $\omega = \mathsf{a}(\,u_1, ..., u_n\,)$ is said to have an environmental alibi if, after applying the substitutions of the enclosing environments, the predicate $\bigvee_{i=1}^{n}(\neg isVariable(u_i) \wedge (u_i \neq v_i))$ is evaluated to TRUE, *where $isVariable(x)$ returns* TRUE *if x is a variable.*

Alibi Example. Let the following definitions: $\sigma = \mathsf{a}(\,1, 2\,)$, $E = |||\, x \in 1..5 \,:\, \mathsf{a}(\,x, 2\,)$, $F = [\![y := 2]\!]\big(|||\, x \in 1..5 \,:\, \mathsf{a}(\,x, y\,)\big)$ and $G = [\![y := 3]\!]\big(|||\, x \in 1..5 \,:\, \mathsf{a}(\,x, y\,)\big)$. If environments are applied as substitutions, then we can rewrite $F = |||\, x \in 1..5 \,:\, \mathsf{a}(\,x, 2\,)$ and $G = |||\, x \in 1..5 \,:\, \mathsf{a}(\,x, 3\,)$. Then E and F are similar. Now the predicate is applied to the action $\mathsf{a}(\,\_, \_\,)$ of each PE. For E and F, x is bounded by the $|||\, x \in 1..5$ operator and the second parameter of a equals the second parameter of σ so there is no environmental alibi for both actions. For G, x is bounded, $2 \neq 3$; hence the action in G has an environmental alibi.

Definition 2. ***Victims relative to σ***
Let E be a PE and σ a query refused by E. Victims relative to σ are defined as the set of leaves of E whose label is the same as σ and does not have environmental alibi.

Example. In the library IS described in Fig. 1, if the user tries to execute the action $\mathsf{DisplayNumberOfLoans}(\,John\,)$ where $John$ is a valid id for a member (*i.e.* $John \in$ MEMBERID) but $John$ is not a registered member (*i.e.* $\mathsf{Join}(\,John\,)$ was never executed) then the system can not execute the query of the user.

Victims relative to $\mathsf{DisplayNumberOfLoans}(\,John\,)$ is the list reduced to one leaf: $\mathsf{DisplayNumberOfLoans}(\,John\,)$ in the **member** process of the initial specification. Fig. 2 show the position of this leaf in the **member** process. In this simplified AST of the **member** entity, some operators are numbered to help further references. Operators that will not be referenced are not numbered.

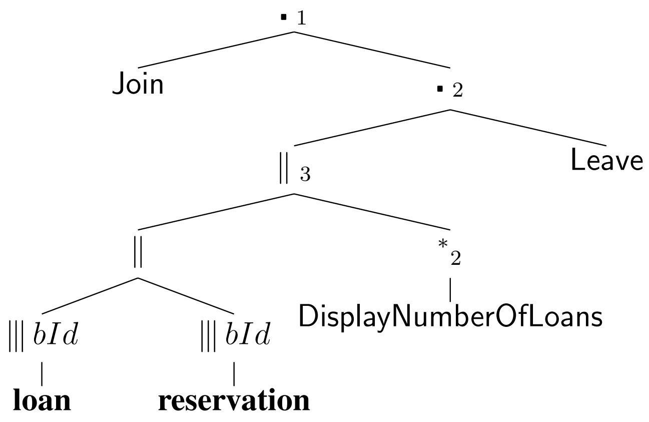


Fig. 2. Simplified AST of **member** entity in the EB^3 specification of the library

3.2 Witnesses

Since the query of the user is refused by the PE, the causes of the error are one or more operators related to the action. Based on the victims, a list of witness operators is established. This list contains the list of all operators on the path from a leaf in the list of the victims to the root of the PE.

Definition 3. ***Witnesses relative to a victim*** v
Let E be a PE, σ a query from the user refused by E and v a victim of E relative to σ. Witnesses relative to a victim v are defined as the list of operators which are parents of v in E.

Example. The calculus of the witnesses relative of a victim is easy to do with the AST representation of the PE. Since only one victim was found relative to the query of the user, only one list of witnesses will be generated by this step.

The list of witnesses relative to the victim of DisplayNumberOfLoans($John$) is: $\|_1$, $\|_2$, $|||$ mId, $^*{}_1$, $\cdot_1$, $\cdot_2$, $\|_3$, $^*{}_2$. The first four operators come from the PE of the **main** process; they are omitted from Fig. 2 for the sake of concision. The last four come from the PE of the **member** process.

3.3 Suspects

To reduce the field of investigation, some operators, which we call *permissive*, are removed from witnesses. Permissive operators can execute the query if their operands can. In other words, if the query fails, it is not their fault, but the fault of the operators occurring in their operands. This is the crux of the permissiveness evaluation described below. The predicate $isPermissive$ is applied to each witness to evaluate if it can be removed (TRUE) or considered as suspect (FALSE). The following algorithm is written using the functional language Caml [9], and uses pattern matching in order to evaluate the predicate $isPermissive$.

Algorithm 2. $isPermissive(F, \sigma, \Gamma)$

Description: *Evaluates if an operator must not be considered as suspect.*
input F**:** *An AST whose root is the witness operator for which permissiveness is evaluated. At least one victim must occur in* F*.*
input σ**:** *A query from the user.*
input Γ**:** *The enclosing environment of* F*.*

```
let $isPermissive(F, \sigma, \Gamma)$ = match $F$ with
    $E_1 \mid E_2$                         $\rightarrow$ TRUE
    $E_1{}^*$                              $\rightarrow$ TRUE
    $E_1 . E_2$                            $\rightarrow label(\sigma) \in alphabet(E_1) \vee \lambda\text{-}terminate(E_1, \Gamma)$
    $\mid x \in X : E_1$                   $\rightarrow param(\sigma)[kappaIndice(F, label(\sigma))] \in X$
    $E_1 |[\Delta]| E_2$                   $\rightarrow label(\sigma) \notin \Delta \vee$
                                           $(label(\sigma) \in alphabet(E_1) \wedge label(\sigma) \in alphabet(E_2))$
    $|[\Delta]| \, x \in X : E_1$          $\rightarrow param(\sigma)[kappaIndice(F, label(\sigma))] \in X$
    $(\![\omega]\!) E_1$                   $\rightarrow$ TRUE
    $P \Longrightarrow E_1$                $\rightarrow evaluate(P, \Gamma)$
```

where $alphabet(E)$ *returns the set of all action labels used in E,* $label(\sigma)$ *returns the label of* σ*,* $param(\sigma)$ *returns the list of the parameters of* σ*,* $kappaIndice(E, l)$ *returns the position of the quantified variable in E for the action whose label is l, and* $evaluate(P, \Gamma)$ *returns the evaluation of the predicate P under the environment* Γ*.* $\lambda\text{-}terminate$ *is described below in the paragraph of sequence operator.*

In the sequel, we illustrate the algorithm for each operator.

Choice. The choice operator is permissive since if one of its operands can execute the query, then the choice can also execute it. For instance, when $\sigma = \mathsf{a}$, the choice operator is permissive in $(\mathsf{b} . \mathsf{a}) \mid \mathsf{b}$.

Kleene closure. This operator is always permissive by definition. Indeed, if E can execute sigma, then E^* can execute sigma. That is why an error is never due to the Kleene closure. When $\sigma = \mathsf{a}$, the Kleene closure is permissive in $(\mathsf{b} . \mathsf{a})^*$.

Sequence. The sequence operator may be permissive depending on the location of the label of σ in its operands. Semantically, there are two cases. In the first case, the action occurs in the left operand; then the sequence operator is permissive, since the left operand should be able to execute the action; hence, the sequence is not responsible for the error. For instance, let $\sigma = \mathsf{a}$ in $(\mathsf{b} ._1 \mathsf{a}) ._2 \mathsf{b}$. The first occurrence $._1$ is not permissive, since a doesn't occur in the left operand; the second occurrence $._2$ is permissive, since a occurs in its left operand.

In the second case of permissiveness, the action occurs in the right operand and the left operand can terminate successfully by executing a sequence of the internal action λ, a condition which we denote by $\lambda\text{-}terminate$. For instance, let $\sigma = \mathsf{a}$ in $\mathsf{b}^* ._1 (\mathsf{b} ._2 \mathsf{a})$. The first occurrence $._1$ is permissive, since b^* can $\lambda\text{-}terminate$, while occurrence $._2$ is not permissive, since b can not $\lambda\text{-}terminate$. For the sake of concision, the definition of λ-termination is omitted; the reader is referred to [4] for a complete definition. It can be computed in $O(n)$, where n is the size of the AST. λ-termination is similar to ϵ-transitions in regular expressions and non-deterministic automata.

Quantified Choice. The quantified choice operator is permissive if the set of the quantification contains the value of the corresponding parameter in the query. Let $\sigma = \mathsf{a}(\,4\,)$ in the following. In $|\, x \in 1..3 \,:\, \mathsf{a}(\,x\,)$, the quantified choice operator is not permissive. Whereas in $|\, x \in 1..4 \,:\, \mathsf{b}(\,x\,) \,.\, \mathsf{a}(\,x\,)$ the quantified choice operator is permissive.

Synchronization. Synchronization operators are permissive on non-synchronized actions. If an action is synchronized then all the operands of the synchronization must contain the action. Otherwise the synchronization is declared not permissive. Let $\sigma = \mathsf{a}$ in the following. For $(\mathsf{b} \,.\, \mathsf{a})|[\emptyset]|\,\mathsf{b}$, $|[\emptyset]|$ is permissive. For $(\mathsf{b} \,.\, \mathsf{c})|[\mathsf{c}]|\,(\mathsf{c} \,.\, \mathsf{a})$, $|[\mathsf{c}]|$ is permissive. For $(\mathsf{b} \,.\, \mathsf{a})|[\mathsf{a}]|\,(\mathsf{a} \,.\, \mathsf{b})$, $|[\mathsf{a}]|$ is permissive. For $\mathsf{b}|[\mathsf{a}]|\,\mathsf{a}$, $|[\mathsf{a}]|$ is not permissive because a is not in both sides of the synchronization despite it is in the synchronization set.

Quantified Synchronization. As for quantified choice, quantified synchronization operator is permissive if the set of the quantification contains the value of the corresponding parameter in the query.

Guard. A guard allows executing the guarded expression if, and only if the guard is evaluated to TRUE. In this case, the guard is permissive. In the other case, the guard is declared as not permissive. Let $\sigma = \mathsf{a}$ in the following. For TRUE $\Longrightarrow (\mathsf{b} \,.\, \mathsf{a})$, the guard is permissive. For $x \geq 0 \Longrightarrow (\mathsf{b} \,.\, \mathsf{a})$ under the environment $[\![x := 1]\!]$, the guard is permissive. For FALSE $\Longrightarrow (\mathsf{b} \,.\, \mathsf{a})$, the guard is not permissive. For $x \geq 10 \Longrightarrow (\mathsf{b} \,.\, \mathsf{a})$ under the environment $[\![x := 1]\!]$, the guard is not permissive. Note that the environment is always permissive; its value determines the permissiveness of enclosed operators.

Establishing Suspects List. After the evaluation of the predicate $isPermissive$ for all the witnesses, the suspects list can be determined.

Definition 4. ***Suspects relative to*** σ
Let E be a PE, σ a query from the user that generates an execution error, v a victim of E relative to σ and w its witnesses. Suspects are defined as the list of operators which are in w and that are not permissive.

Example. The predicate $isPermissive$ is applied on each member of the witnesses list. The operator $._1$ is not permissive because σ is located in the right whereas for the operator $._2$, σ is on the left as it can be seen at Fig. 2. All others operators are permissive as shown in Table 1.

3.4 Culprits

For each list of suspects (*i.e.* one list per witness), the head (the nearest operator to the root) is declared guilty. In case this list is empty, no operator is declared guilty, and the

Table 1. Suspects relative to the victim of DisplayNumberOfLoans($John$)

Witness	$isPermissive$	Why ?	Suspect
$\|\|_1$	TRUE	Synchronization ok	no
$\|\|_2$	TRUE	Synchronization ok	no
$\|\|\| \, mId \in$ MEMBERID :	TRUE	$John \in$ MEMBERID	no
$^*{}_1$	TRUE	Always permissive	no
$\cdot_1$	FALSE	Right operand and no λ-termination	yes
$\cdot_2$	TRUE	Left operand	no
$\|\|_3$	TRUE	Synchronization ok	no
$^*{}_2$	TRUE	Always permissive	no

algorithm can not provide an error message for the request of the user. In the other case, a list of culprit operators is established and this concludes the investigations to find the cause of an execution error.

Definition 5. ***Culprit relative to a victim***
Let E be a PE, σ a query from the user refused by E, v a victim of E for σ and s its suspects. The culprit relative to v is defined as the operator which is in s and that is the closest to the root.

Example. Since only one suspect was found for DisplayNumberOfLoans($John$), this operator is automatically declared guilty.

4 Generating Messages for an Execution Error

After the localization of the cause of an error, the IS must build its error messages in order to explain to the user the cause of the failure to execute his query. This process is divided in several steps, as the message is built for several goals.

4.1 Messages from Culprits

For each culprit found, a message is generated according to the type of operator. This message is built using a skeleton that must be completed by information from several origins, like the query, parameters of the query, and elements from the culprit operator (if any). This message aims to report the cause of the error and its context to the user with related information.

Message Skeletons by Operators. This algorithm uses pattern matching to link operators to message skeletons. Operators which are always permissive are not considered in the algorithm, since they can never be culprits.

Algorithm 3. $skeletonFromCulprit(\sigma, culprit)$

Description: *Returns the message skeleton for a culprit found for σ.*
input σ**:** *A query from the user.*
input $culprit$**:** *An operator that was previously declared as culprit.*

```
let $skeletonFromCulprit(\sigma, culprit)$ = match $culprit$ with
    $E_1 \,.\, E_2$                        $\rightarrow$ "Execution order of actions is not respected."
    $|\, x \in \mathrm{X} : E_1$            $\rightarrow$ "The $x$ variable of the query $\sigma$ is not in X."
    $E_1 |[\Delta]| E_2$                 $\rightarrow$ "The system will not be able to execute $\sigma$ in this process."
    $|[\Delta]|\; x \in \mathrm{X} : E_1$   $\rightarrow$ "The $x$ variable of the query $\sigma$ is not in X."
    $P \Longrightarrow E_1$              $\rightarrow$ "The constraint $P$ is not verified."
```

Example. For the victim DisplayNumberOfLoans($John$), the culprit operator is $\,._1$. Then the skeleton message associated to this operator is "*Execution order of actions is not respected.*".

4.2 Required Action

In order to help the user after the display of the error message, the system must invite him to do something. That is why the algorithm computes *required actions*. These actions are submitted as advice to help the user to execute his initial query.

Definition 6. ***Required action***
Let E be a PE, σ a query from the user refused by E, v a victim of E for σ and c the culprit associated with v. A required action associated to c to execute σ in E is an action that must be executed before considering to execute σ.

After executing a required action, the user can not assume that σ will be executed without any error. Other attempts to execute σ may fail again, but these attempts may generate other error messages.

Algorithm. Since there can be several required actions for a single culprit, the algorithm stocks them into a list. The algorithm also takes into account the fact that some actions may not be always executable due to guards or other conditions. That is why the algorithm uses a pair (action, Boolean) to represent a required action. In the case that the Boolean is FALSE, the action can not be executed in the current state of the IS, but is still required in order to execute σ.

Algorithm 4. $requiredA(exp, \sigma, \Gamma, \phi)$

Description: *Returns a list of required actions for a culprit found for σ.*
input exp**:** *An operator (PE) that is initially a culprit of σ.*
input σ**:** *A query from the user.*
input Γ**:** *An environment of the PE exp.*
input ϕ**:** *A condition associated with the required action, initially* TRUE.

```
let rec $requiredA(exp, \sigma, \Gamma, \phi)$ =match $exp$ with
    $Action(label, \_)$          $\rightarrow [(exp, \phi)]$
    $E_1 \mid E_2$               $\rightarrow requiredA(E_1, \sigma, \Gamma, \phi) \oplus requiredA(E_2, \sigma, \Gamma, \phi)$
    $E^*$                        $\rightarrow requiredA(E, \sigma, \Gamma, \phi)$
    $E_1 \,.\, E_2$              $\rightarrow requiredA(E_1, \sigma, \Gamma, \phi)$
    $\mid x \in \mathrm{X} : E_1$       $\rightarrow requiredA(E_1, \sigma, \Gamma, \phi)$
    $E_1 \mid[\Delta]\mid E_2$   $\rightarrow requiredA(E_1, \sigma, \Gamma, \phi) \oplus requiredA(E_2, \sigma, \Gamma, \phi)$
    $\mid[\Delta]\mid x \in \mathrm{X} : E_1$ $\rightarrow requiredA(E_1, \sigma, \Gamma, \phi)$
    $(\!|\omega|\!) E_1$         $\rightarrow requiredA(E_1, \sigma, (\Gamma \lhd \omega), \phi)$
    $P \Longrightarrow E_1$      $\rightarrow requiredA(E_1, \sigma, \Gamma, \phi \wedge P)$
```

where $\oplus$ is the classical concatenation operator for lists and $\lhd$ denotes the environment composition.

Translation to a Recommendation. Since the user can only execute one action at the same time, only one required action will be executed just after the broadcast of the error message, or none if the user does not take this advice into account. When a required action $(\omega\ , \phi)$ is computed, then the recommendation addressed to the user is: "*In order to execute σ, you should try executing ω under the condition ϕ.*"

4.3 Information System Patterns

An IS generated using the EB^3 method is produced from (among other things) an entity-relationship diagram. Several PE patterns based on this diagram are defined in [2]. These patterns may be used to deduce an improved version of the error message.

Definition 7. ***Entity pattern***
Let $\mathsf{P}(x)$, $\mathsf{M}(x)$ and $\mathsf{C}(x)$ be PEs and let E be a PE that matches the following pattern, where $\mathsf{P}(x)$ denotes a choice of producers, $\mathsf{M}(x)$ denotes a choice of modifiers, $\mathsf{C}(x)$ denotes a choice of consumers and x a key for an entity with x in XID *:*

$$|||\, x \in \mathrm{XID} : \mathsf{P}(x) \,.\, \mathsf{M}(x)^* \,.\, \mathsf{C}(x)$$

In the case that the entity pattern is a sub-expression of the current state of an IS and σ, the query of the user, is a modifier or a consumer relative to this entity pattern, if the system can not execute σ, then the error message skeleton associated to the . operator can be upgraded to: "*The entity x of type E does not exist.*" The translation to a recommendation of the required action can be more precise: "*In order to create it, you should try executing ω under the condition ϕ.*" with $(\omega\ , \phi)$ as computed by the $requiredA$ algorithm.

Definition 8. 1-n ***relationship pattern***

Let $\mathsf{P}_1(x)$ and $\mathsf{C}_1(x)$ be PEs and let E_1 be a PE that matches the following pattern:

$$E_1 = |||\, x \in \mathrm{XID} : \mathsf{P}_1(x) \,.\, (|||\, y \in \mathrm{YID} : \mathsf{A}(x, y))^* \,.\, \mathsf{C}_1(x)$$

Let $\mathsf{P}_2(y)$ and $\mathsf{C}_2(y)$ be PEs and let E_2 be a PE that matches the following pattern:

$$E_2 = |||\, y \in \mathrm{YID} : \mathsf{P}_2(y) \,.\, (|\, x \in \mathrm{XID} : \mathsf{A}(x, y))^* \,.\, \mathsf{C}_2(y)$$

Let $F = E_1 \parallel E_2$, then F is said to follow the 1-n relationship pattern.

In the case that the 1-n relationship pattern is a sub-expression of the current state of an IS and σ, the query of the user, is an action of A, if the system can not execute σ, then the error message skeleton associated to one of the $\centerdot$ operators of E_1 or E_2 can be upgraded to : "*The relationship* A *between the entity* x *of type* E_1 *and the entity* y *of type* E_2 *does not exist. You can create it by executing* ω*.*" with $(\omega\ , \phi)$ as computed by the $requiredA$ algorithm. This message may even become more pertinent with a label name associated to both entities and the 1-n relationship. As a good practice, relationship names should be used as process names in the formal specification.

The n-n relationship pattern is similar to the 1-n relationship pattern except that the E_2 PE matches the following pattern:

$$||| \, y \in yId \, : \, \mathsf{P}_2(\, y \,) \centerdot (||| \, x \in xId \, : \, \mathsf{F}(\, x, y \,))^* \centerdot \mathsf{C}_2(\, y \,)$$

Example. In our case, the culprit operator is $\centerdot_1$ and above it there is a quantified interleaving in the list of witnesses. This is the entity pattern, so the skeleton message of $\centerdot_1$ can be replaced. The required action algorithm applied to our culprit generate only one element in the list: $(\mathsf{Join}(\, mId \,)$, TRUE$)$. Then we can build the full error message that will be displayed: "*The '*$member$*' entity '*$John$*' does not exist. In order to create it, you should try executing* $\mathsf{Join}(\, mId \,)$*.*"

5 Case Study

This section will apply the process described above to other errors.

5.1 Current State

Since a system evolves from its initial state, and in order to illustrate some more complex cases of execution error, a state, different from the initial state (see Fig. 1) is used in the following cases.

First, BOOKID is defined as the set of naturals between 0 and 9, MEMBERID is defined as the set of naturals between 10 and 19. Then, the library is populated with 2 books (with bId 0 and 1) and 3 members (with mId 10, 11 and 12). To finish, member 10 has borrowed book 0, just returned book 1 and members 11 and 12 have made reservation for book 1 in this order.

In short, the current state used below is the state resulting from the execution of the actions $\mathsf{Acquire}(\, 0 \,)$, $\mathsf{Acquire}(\, 1 \,)$, $\mathsf{Join}(\, 10 \,)$, $\mathsf{Join}(\, 11 \,)$, $\mathsf{Join}(\, 12 \,)$, $\mathsf{Lend}(\, 10, 0 \,)$, $\mathsf{Lend}(\, 10, 1 \,)$, $\mathsf{Reserve}(\, 11, 1 \,)$, $\mathsf{Reserve}(\, 12, 1 \,)$ and $\mathsf{Return}(\, 10, 1 \,)$ in this order upon the initial specification.

5.2 Generated Messages

Trying to Borrow a Currently Borrowed Book. If a member wants to borrow a book that is already borrowed by another member, the system should refuse the query and warn the user about this error. The user could expect from a system with "human-generated" error messages a message that explains to him that the book is not available.

Applying the "police investigation" method on this state while attempting to execute Lend(11, 0) produces a list of seven victims (with six environmental alibis found). Since several victims can share the same culprit, fewer messages will be generated. Only one victim is matching the exact query from the user for both environments and quantification sets. Here is the complete execution of the algorithm for this victim.

Table 2. Investigation over Lend(11, 0)

Witnesses	Suspect	Why ?	Culprit	Pattern
$\vert\vert\vert$	no	Always permissive	-	-
$(\![bId := 0]\!)$	no	Always permissive	-	-
$\cdot$	yes	Right operand and no λ-termination	yes	-
$\vert\, mId \in [10, 19]\ :$	no	$mId \in$ [10,19]	-	-

From this culprit, the algorithm provides a skeleton message that is used to generate the first part of the message: "*Execution order of actions is not respected.*"

Then, in addition to this first part are generated required actions for this culprit. The execution of the algorithm returns the action Return(10, 0) with condition TRUE that we can translate into: "*In order to execute* Lend*(* 11, 0 *), you should try executing* Return*(* 10, 0 *).*" Since no known pattern is detected in this investigation, the final message is generated: "*Execution order of actions is not respected. In order to execute* Lend*(* 11, 0 *), you should try executing* Return*(* 10, 0 *).*"

Trying to Bypass the Reservation List. In case that the user of the system tries to execute an action protected by a FALSE-evaluated guard, the system must block the execution of the query. In the library, this can happen when a user wants to bypass the reservation process by taking a book while he is not the first on the reservation list.

Trying to execute Take(12, 1) in the current state where member 11 is first in the reservation list produces the investigation shown in the Table 3.

Table 3. Investigation over Take(12, 1)

Witnesses	Suspect	Why ?	Culprit	Pattern
$\vert\vert$	no	Synchronization ok	-	-
$\vert\vert\vert$	no	Always permissive	-	-
$(\![mId := 12]\!)$	no	Always permissive	-	-
$\vert\vert$	no	Synchronization ok	-	-
$\vert\vert\vert$	no	Always permissive	-	-
$(\![bId := 1]\!)$	no	Always permissive	-	-
$\cdot$	no	Left operand	-	-
$isFirst(trace, mId, bId) \Longrightarrow$	yes	FALSE-evaluated	yes	-

In this case, there are no required action before executing Take($12,1$), the guard is the only thing that prevents the user to execute the query: "*In order to execute* Take*($12,1$), the predicate $isFirst(trace, mId, bId)$ must be evaluated to* TRUE*.*"

It could be interesting to point to the user of the IS exactly what makes the guard evaluated to FALSE. In the library specification, the predicate $isFirst(trace, mId, bId)$ is specified as a recursive function over the trace of the IS. The trace of the system consists of a sequence of all the actions executed since the launch of the system. In our example, we could provide to the user the name of the member that is the first in the reservation list. On the other hand, this raises the issue of confidentiality; hence, there is a need for configurability when selecting messages to display. Another analysis that could be done is to determine, from the definition of the function $isFirst$, which actions can make the guard true. This means solving a predicate, which is a hard problem in the general case, but simple heuristics could probably solve a large number of simple cases.

6 Conclusion

We have presented an algorithm that can generate an error message for an event refused by a process expression. The algorithm is generic, in that it can be used on any process expression. The algorithm identifies the causes of an error and suggests corrective actions. Currently, nothing can ensure that a required action will be executable. This limitation may be annoying for the user of the system since he can not receive useful advice. But if he tries to execute the required action, he can receive another error message that will help him to execute his initial request.

In the case of non-fulfilled guards, the user may not be able to know what to do in order to comply with a guard. Indeed, nothing indicates how to modify the state of the system or the value of attributes that are involved in the guard. By finding which attributes are involved in a guard and providing a list of actions that modify these attributes, the error message could help the user in order to satisfy a guard.

Another limitation could be the names of the variables and actions of the PE. Indeed, without a pertinent name, error message will integrate impertinent information that will not be easy to understand by the user. That is why this method requires an active involvement of the designer of the system by using pertinent and understandable names in his specification.

The message skeletons we have provided are rather simple. They can be enhanced; for instance, we can compute the query parameter associated to the quantification variable mentioned in a skeleton. We could use this parameter name instead (each action has a signature with parameter names). One can also imagine that annotations could be added to the specification which could automatically be included in the skeletons to provide more information. Guard predicates could also be translated into a natural language representation. They could also be further analysed to target specific subpredicates which make the overall predicate false.

To improve readability, a selection of one among all the generated messages according to a heuristic could be interesting. This heuristic could vary depending on the target and the goal of the message. However, this feature has not been developed for the moment.

The approach of automatic deduction of pertinent error messages from EB^3 specification could be adapted to other specification languages such as CSP [7]. Indeed, the method to find the cause of an error could be transposed to another process algebra, with some modifications to the "permissive" and the "message skeleton" parts.

CSP operators for sequence "$\rightarrow$" and ";" are similar to EB^3 operator "**.**". Hence, the permissiveness and the message skeleton of these operators would not change. CSP external choice operator "$\Box$" is also similar to EB^3 "$|$" whereas CSP's internal choice "$\sqcap$" has no equivalent in EB^3. This operator would need a new predicate to evaluate its permissiveness and a new message skeleton.

Acknowledgements. The authors would to thank the anonymous referees for their insightful comments. This research is supported by NSERC (Natural Sciences and Engineering Research Council of Canada).

References

1. Fraikin, B., Gervais, F., Frappier, M., Laleau, R., Richard, M.: Synthesizing information systems: the APIS project. In: Rolland, C., Pastor, O., Cavarero, J.L. (eds.) First International Conference on Research Challenges in Information Science (RCIS), Ouarzazate, Morocco, April 2007, vol. 12 (2007)
2. Frappier, M., St-Denis, R.: EB^3: an entity-based black-box specification method for information systems. Software and Systems Modeling 2(2), 134–149 (2003)
3. Fraikin, B., Frappier, M., Laleau, R.: State-based versus event-based specifications for information system specification: a comparison of b and eb3. Software and System Modeling 4(3), 236–257 (2005)
4. Fraikin, B.: Interprétation efficace d'expression de processus EB^3. PhD thesis, Département d'informatique, Université de Sherbrooke, Sherbrooke, Québec, Canada (April 2006)
5. Brown, P.J.: Error messages: the neglected area of the man/machine interface. Communications of the ACM 26(4), 246–249 (1983)
6. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Reading, Mass (1986)
7. Hoare, C.A.R.: CSP–Communicating Sequential Processes. Prentice Hall, Englewood Cliffs (1985)
8. Milhau, J., Fraikin, B.: Technical report 25, an algorithm for automatic generation of error messages for EB^3. Technical report, Université de Sherbrooke, Département d'informatique (September 2008)
9. Leroy, X., Weis, P.: Manuel de référence du langage Caml. InterEditions (1993)

Decompositional Petri Net Reductions[⋆]

Astrid Rakow

Universität Oldenburg

Abstract. As a means to tackle the state explosion problem of model checking 1-safe Petri nets for linear time logic without next-time (LTL-$_{\mathrm{X}}$), an approach that combines compositional verification and Petri net reductions is presented. We decompose a Petri net into (i) a so-called kernel net Σ_{k} containing all places mentioned by the LTL-$_{\mathrm{X}}$ property ϕ and (ii) environment subnets $\Sigma_{\mathrm{e}_1}, ..., \Sigma_{\mathrm{e}_n}$. These environment nets do not interact with each other and have limited influence on the kernel only. Six distinct and very simple summary nets suffice to describe the influence of any environment net. To determine the appropriate summary net $S(\Sigma_{\mathrm{e}_i})$ we modularly verify up to three fixed LTL-$_{\mathrm{X}}$ formulas on Σ_{e_i}. We reduce Σ by replacing every environment subnet Σ_{e_i} in Σ by its summary net. Instead of checking ϕ on Σ, we check ϕ on the reduced net. Verification of several case-studies shows that our reduction approach can significantly speed-up model checking.

1 Introduction

Model checking is a technique to automatically verify that a system satisfies a formal property by exploring the system's state space. The main challenge in model checking is that the number of states may grow exponentially in the system size, which is widely known as state explosion. Among others compositional verification and Petri net reductions are two techniques to combat the state explosion problem for model checking Petri nets. Compositional verification allows to infer global properties from verification of the system's components. By verifying the components in isolation, the combinatorial blow-up of the state space is avoided and verification is possible at much lower computational expense. Principal challenges of using compositional verification for monolithic Petri nets are to find an appropriate decomposition and local properties to check on the components. A major issue in determining a decomposition for a monolithic Petri net is finding the right border, since it is possible that a component exposes spurious behaviour in isolation, i.e., behaviour that the component does not have as part of the global system due to context constraints imposed by the component's environment.

Petri net reduction is a method to reduce the Petri net graph (as opposed to the often considered reachability graph) while preserving some properties of

[⋆] This work is supported by the German Research Foundation (DFG), grant GRK 1076/1.

interest. Petri net reductions for model checking aim to reduce a Petri net graph such that the reduced net has a smaller state space.

In this paper we present a new approach that combines compositional verification and Petri net reduction. We decompose a monolithic 1-safe Petri net Σ for a given LTL-$_\text{X}$ (LTL without next-time) property into a kernel net Σ_k and environment nets $\Sigma_{\text{e}_1}, ..., \Sigma_{\text{e}_n}$. The kernel subnet contains all places mentioned by the LTL-$_\text{X}$ property ϕ and shares with an environment net a single place q only. Every environment net Σ_{e_i} is replaced by a summary net $S(\Sigma_{\text{e}_i})$, which is a small net of at most four nodes and describes Σ_{e_i}'s influence on the kernel precisely. To determine the appropriate summary net $S(\Sigma_{\text{e}_i})$, Σ_{e_i} is model checked independently to characterise its influence on the kernel. For this up to three local and fixed LTL-$_\text{X}$ properties are checked on Σ_{e_i}. We establish whether ϕ holds on Σ by model checking whether ϕ holds on the reduced net. The decomposition of a monolithic 1-safe Petri net can be determined in linear time, and by choosing a restrictive interface between kernel and environment, that is the single place q, the risk of encountering spurious behaviour within an isolated environment net is minimised.

Related Work. A decompositional approach to check boundedness and liveness of a monolithic Petri net is described by Lee et al. in [1]. A net is decomposed based on minimal linear invariants. The generated components may overlap, that is share places and transitions. The components' reachability graphs are reduced and (re)composed to analyse the global behaviour, where shared transitions have to be synchronised. An iterative approach to decompose a monolithic Petri net for checking LTL-$_\text{X}$ properties is presented by Klai et al. in [2]. The generated components approximate the global behaviour. If a component does not satisfy the property under consideration, the validity of the counterexample has to be checked by means of a so-called non-constraining relation, which represents the environment's constraints. If this relation is not satisfied, the net is reexamined under a coarser partition. In the approach of Lee et al. spurious behaviour is ruled out by synchronising the transitions shared among components. In the approach of Klai et al. spurious behaviour may lead to repartition such that the global behaviour is captured more accurately with each iteration. We will show that our decomposition allows us to accurately characterise the influence of the environment net on the kernel net, so that neither local components need to be synchronised as in the approach of [1] nor an iterative refinement as in [2] is necessary. When model checking an environment net Σ_{e_i} to characterise its influence on the kernel, only in one out of six cases is it possible that Σ_{e_i} exposes spurious behaviour. As we see model checking as one of the later steps taken when analysing a Petri net, we argue that often this case might be ruled out by earlier results of the analysis.

Valmari suggested in [3] an approach to compositional analysis for Petri nets whose subnets share a set of places only. It is assumed that the given net is divided into an environment component and an interesting component, i.e. a kernel. A labelled transition system representing the environment net's behaviour is condensed by a CFFD-semantics preserving algorithm. The environment net is then replaced by a net corresponding to the condensed labelled transition system. Since in our approach kernel and environment net share a single place only, we consider a special case of the scenario examined in [3]. In our case we can identify six fixed summary nets that suffice to describe the influence of any environment net, whereas in [3] the replacement net is the result of a condensation. Also we use model checking to determine the replacement and thus can make use of the various methods that have been invented to speed up model checking.

Several works on Petri net reductions that preserve some properties of interest have been suggested, for instance [4,5]. Petri net reductions for model checking have been examined in [6,7,8]. Poitrenaud and Pradat-Peyre showed in [6] that the local net reduction rules called Pre- and Post-agglomeration of Berthelot [4] preserve LTL$_{\text{-X}}$ properties. In [7] Esparza and Schröter presented a set of reduction rules for LTL$_{\text{-X}}$ model checking of 1-safe Petri nets based on invariants, implicit places and local net reductions, thereby extending the works of [4,6]. In [8] Haddad and Pradat-Peyre further generalised the Pre- and Post-agglomeration rules by defining behavioural preconditions and sufficient structural conditions based on the description of linear programs. Whereas these reductions all focus on replacing a fixed structure, e.g. a transition, input and output places, we introduce here a reduction rule that replaces a variable subnet.

Outline of the Paper. In Sect. 3 we introduce the LTL$_{\text{-X}}$ formulas that are model checked on an environment net Σ_{e_i} in order to characterise its influence on the kernel and we present the six distinct summary nets that suffice to equivalently describe Σ_{e_i}'s influence. A net reduced by replacing Σ_{e_i} in Σ by the summary net $S(\Sigma_{\mathrm{e}_i})$ satisfies an LTL$_{\text{-X}}$ property ϕ if and only if Σ satisfies ϕ, provided we can assume a very weak form of fairness that guarantees some progress on Σ_{k}. The correctness results of our reductions are also given in Sect. 3. Section 4 illustrates the decomposition algorithm for 1-safe Petri nets and discusses the computational expense of determining a decomposition. In Sect. 5 we show that our approach can efficiently speed up the model checking time and examine a combination of our approach and the partial order approach of stubborn sets [9]. We conclude in Sect. 6 with a discussion of our approach. In the next section we introduce the basic notions of this paper.

2 Preliminaries

Petri Net Definitions. A *Petri net* N is a triple (P, T, W) where P and T are disjoint sets and $W : ((P \times T) \cup (T \times P)) \to \mathbb{N}$. An element $p \in P$ is called a *place* and $t \in T$ a *transition*. The function W defines weighted arcs between places and transitions. We denote the restriction of W to $((P' \times T') \cup (T' \times P'))$ as $W|_{(P',T')}$ for $P' \subseteq P$, $T' \subseteq T$. The *preset* of $t \in T$ is ${}^\bullet t = \{p \in P \mid W(p,t) > 0\}$, its *postset* is $t^\bullet = \{p \in P \mid W(t,p) > 0\}$. Analogously ${}^\bullet p$ and $p^\bullet$ are defined.

A *marking* of a net N is a function $M : P \to \mathbb{N}$, which assigns a number of tokens to each place. $M|_{P'}$ is the restriction of M to places $P' \subseteq P$. With a given order on the places, $p_1, ..., p_n$, M can be represented as a vector in $\mathbb{N}^{|P|}$, where the i-th component is $M(p_i)$. As $M^{q=x}$ we denote the marking that places x tokens on q and $M(p)$ tokens on any other place p.

A transition $t \in T$ is *enabled* at marking M, $M[t\rangle$, iff $\forall p \in {}^\bullet t : M(p) \geq W(p,t)$. If t is enabled it can *fire*. The firing of t generates a new marking M', $M[t\rangle M'$, which is determined by the firing rule as $M'(p) = M(p) + W(t,p) - W(p,t), \forall p \in P$. The definition of $[\rangle$ is extended to transition sequences σ as follows. A marking M always enables the empty firing sequence ε and its firing generates M. M enables a transition sequence σt, $M[\sigma t\rangle$, iff $M[\sigma\rangle M'$ and $M'[t\rangle$. If $M[\sigma\rangle$, the transition sequence σ is called a *firing sequence* from M. A firing sequence σ from M is maximal iff either σ is infinite or σ cannot be extended, i.e., $\neg M[\sigma t\rangle$, $\forall t \in T$. Given a firing sequence $\sigma = t_1 t_2 ...$ with $M_0[t_1\rangle M_1[t_2\rangle M_2 ...$, the sequence $M_0 M_1 M_2 ...$ is called the *marking sequence* from M_0, $\mathcal{M}(M_0, \sigma)$. A marking sequence $\mathcal{M}(M, \sigma)$ is *maximal* iff σ is a maximal firing sequence. By convention, we regard a finite maximal marking sequence μ as equivalent to the infinite marking sequence μ' that repeats the final marking of μ infinitely often.

We denote $X^* \cup X^\omega$ as X^∞ for a set X. For a finite sequence $\gamma = x_1 x_2 ... x_n \in X^\infty$, $|\gamma|$ is n, the length of γ. If γ is infinite, $|\gamma| = \infty$. $\gamma(i)$ denotes the i-th element and γ^i denotes the suffix of γ that truncates the first i positions of γ.

A Petri net $\Sigma = (N, M_0)$ with a designated initial marking M_0 is called a *marked Petri net*. A marking of Σ is *reachable* if there is a firing sequence from M_0 that generates M, $M_0[\sigma\rangle M$. The *set of reachable markings* of Σ is denoted as $[M_0\rangle$. A place p is *k-bounded* if any reachable marking has at most k tokens at p. Σ is k-bounded if all of its places are. 1-boundedness is also referred to as *1-safeness*. In the following we use N synonymous with its defining triple (P, T, W) and Σ synonymous with (N, M_0). Also subscripts carry over to components, e.g. $\Sigma_\text{e} = (N_\text{e}, M_{\text{e}0}) = (P_\text{e}, T_\text{e}, W_\text{e}, M_{\text{e}0})$.

Logics. We now briefly introduce the temporal logics LTL and define when an infinite sequence of markings satisfies an LTL formula ϕ. In the next paragraph (cf. Petri Net Semantics) we will define when a Petri Net satisfies ϕ.

Definition 1 (LTL, LTL$_{\text{-X}}$). *Let $AP \subseteq P \times \mathbb{N}$ be the set of atomic propositions. Let $\mu = M_1 M_2 ... \in (\mathbb{N}^{|P|})^\omega$ be an infinite sequence of markings.*

LTL Syntax:
Every $(s, x) \in AP$ is an LTL formula. If ϕ_1 and ϕ_2 are LTL formulas, $\neg\phi_1$, $\phi_1 \wedge \phi_2$, $\mathbf{X}\phi_1$, $\phi_1 \mathbf{U} \phi_2$ are LTL formulas.

LTL Semantics:

$\mu \models (p, x)$	*iff*	$(\mu(1))(p) = M_1(p) = x$
$\mu \models \neg\phi_1$	*iff*	*not* $\mu \models \phi_1$
$\mu \models \phi_1 \wedge \phi_2$	*iff*	$\mu \models \phi_1$ *and* $\mu \models \phi_2$
$\mu \models \mathbf{X}\phi_1$	*iff*	$\mu^1 \models \phi_1$
$\mu \models \phi_1 \mathbf{U} \phi_2$	*iff*	$\exists i, 0 \leq i : \mu^i \models \phi_2 \wedge \forall j, 0 \leq j < i : \mu^j \models \phi_1$

An LTL$_{\text{-X}}$ *formula is an LTL formula built without using the* $\mathbf{X}$ *operator.*

We use the following abbreviations: $\phi_1 \vee \phi_2 \equiv \neg(\neg\phi_1 \wedge \neg\phi_2)$, $true \equiv (p, 1) \vee \neg(p, 1)$, $\Diamond\phi \equiv true\,\mathbf{U}\phi$, $\Box\phi \equiv \neg\Diamond(\neg\phi)$.

We use the function *scope* to associate with every LTL formula ϕ the set of places referred to in ϕ. For example $scope((p_1, 1)\mathbf{U}(p_2, 2))$ is $\{p_1, p_2\}$.

Petri Net Semantics. In the following we define when a Petri net Σ satisfies a formula. Intuitively, a Petri net satisfies a formula ϕ if its behaviour satisfies ϕ. In the sequel we study maximal and fair firing sequences: We consider firing sequences of original net Σ that are fair w.r.t. T_k and maximal firing sequences of Σ_{e_i} – where Σ is composed of environment nets Σ_{e_i} and a kernel Σ_k containing $scope(\phi)$. We assume that Σ is fair w.r.t. T_k, such that the kernel within Σ does not starve, i.e. as long as there is a transition t permanently waiting to be fired, Σ_k eventually fires some transition. Next we define formally that a firing sequence is fair w.r.t. $T' \subseteq T$ (cf. Def. 3), if in case a transition $t \in T'$ is eventually permanently enabled, it fires infinitely often some transition of T'.

Definition 2 (eventually permanently enabled)
Let $\sigma = t_1t_2...$ be a firing sequence of Σ with $M_i[t_{i+1}\rangle M_{i+1}, \forall i,\ 0 \leq i < |\sigma|$.
σ eventually permanently enables $t \in T$ iff
either σ is finite and $M_{|\sigma|}[t\rangle$ or σ is infinite and $\exists i,\ 0 \leq i : \forall j,\ i \leq j : M_j[t\rangle$.

Definition 3 (fairness with respect to T')
Let T' be a subset of T, let $\sigma = t_1t_2t_3...$ be a maximal firing sequence of Σ and M_i be the markings with $M_i[t_{i+1}\rangle M_{i+1}, \forall i,\ 0 \leq i < |\sigma|$.
σ is fair w.r.t. T' iff
– either σ is finite
– or σ is infinite, and, if there is a $t \in T'$ it permanently enables, it then fires infinitely often some transition of T' (which may or may not be t itself).

We also say Σ *is fair w.r.t.* T' to express that we only consider firing sequences of Σ that are fair w.r.t. T'.

Given $scope(\phi)$ is a subset of P, we say Σ *models* ϕ, $\Sigma \models \phi$, iff all maximal firing sequences σ satisfy $\mathcal{M}(M_0, \sigma) \models \phi$. Given a set of transitions $T' \subseteq T$, we say $\Sigma \models \phi$ *fairly w.r.t* T' iff all firing sequences σ that are fair w.r.t. T' satisfy $\mathcal{M}(M_0, \sigma) \models \phi$.

3 The Reduction Rules

In the following we show how to reduce a net Σ composed of a kernel net Σ_k and an environment net Σ_e. An algorithm to determine an appropriate decomposition for a 1-safe net is presented in the next section. Note, that in this section we do not assume that Σ is 1-safe but only require that the shared place q is 1-safe.

Any subnet Σ_e of Σ that shares just a 1-safe place q with the remainder is reducible by our approach. We will show how such an environment net can be summarised by a simple net $S(\Sigma_\mathrm{e})$ and that the environment net can be examined independently to determine its summary, $S(\Sigma_\mathrm{e})$. The restrictive interface, that is the 1-safe place q, between Σ_k and Σ_e allows us to abstract efficiently from the detailed behaviour of Σ_e. Before we show how to reduce a Σ that is composed of Σ_k and Σ_e, we formally define how these two nets compose Σ.

Definition 4 (reducible, kernel, environment)
Let Σ be a marked Petri net and ϕ be an LTL-$_X$ formula. Σ is reducible *by N_e iff there is a 1-safe place $q \in P$ and a subnet N_k, such that $N = (P_\mathrm{k} \uplus (P_\mathrm{e} \setminus \{q\}), T_\mathrm{k} \uplus T_\mathrm{e}, W|_{(P_\mathrm{k},T_\mathrm{k})} \uplus W|_{(P_\mathrm{e},T_\mathrm{e})})$, scope$(\phi) \subseteq (P_\mathrm{k} \setminus \{q\})$ and $q \in P_\mathrm{k}$. Σ is* reducible by *$\Sigma_\mathrm{e} = (N_\mathrm{e}, M_0|_{P_\mathrm{e}})$ iff Σ is reducible by N_e.*

We call $\Sigma_\mathrm{k} = (N_\mathrm{k}, M_0|_{P_\mathrm{k}})$ the kernel *subnet and Σ_e the* environment *subnet.*

So Σ is reducible by an environment net N_e iff Σ is composed of an environment net N_e and a kernel N_k, such that (i) ϕ does not refer to the environment N_e and (ii) kernel N_k and environment N_e have only a 1-safe place q in common, the transitions of kernel and environment are disjoint and they have neither input- nor output places in the other net with exception of q.

In the sequel let ϕ be an LTL-$_X$ formula, let $\Sigma_\mathrm{k} = (N_\mathrm{k}, M_0|_{P_\mathrm{k}})$ be the kernel and $\Sigma_\mathrm{e} = (N_\mathrm{e}, M_0|_{P_\mathrm{e}})$ be the environment subnet of Σ, such that Σ is reducible by Σ_e according to Def. 4. Let q be the place shared by Σ_k and Σ_e. Next we intuitively, then formally introduce the rules to reduce Σ by Σ_e. The reductions are applied to examples in Fig. 1.

Let us assume that Σ is fair w.r.t. T_k, which means that any firing sequence that permanently enables a $t_k \in T_\mathrm{k}$ fires at least one transition of T_k infinitely often. This fairness assumption guarantees progress in Σ_k, since as long as there are transitions in T_k permanently enabled, some transition in T_k is fired. To characterise how Σ_e may affect the given property ϕ we study Σ_e's effect on the 1-safe place q at the two scenarios, Σ_e *with a token on* q, which is $\Sigma_\mathrm{e}^{q=1} = (N_\mathrm{e}, M_0^{q=1}|_{P_\mathrm{e}})$, and Σ_e *without a token on* q, denoted as $\Sigma_\mathrm{e}^{q=0} = (N_\mathrm{e}, M_0^{q=0}|_{P_\mathrm{e}})$.

An environment subnet Σ_e is called a *Borrower* if it may take a token from q -one or several times- but eventually permanently marks q. As we study stuttering-invariant properties, which do not count execution steps [10], Borrower subnets can be omitted.

An environment subnet Σ_e is a *Consumer*, if Σ_e may not return the token from q, i.e. $\Sigma_\mathrm{e}^{q=1}$ has at least one execution that does not eventually *permanently* mark q. Due to our weak fairness notion, progress in Σ_k is only guaranteed, if a transition is eventually permanently enabled, i.e. its preset is permanently (sufficiently) marked. The Consumer subnet Σ_e in Fig. 1 does not eventually permanently mark q. Assume that t_3 has been fired. It is not guaranteed that eventually t_4 is fired. The token may get lost in Σ_e by firing infinitely often $t_2 t_1$. So a Consumer net can be replaced by just one transition that may remove the token from q, just as the Consumer may remove the token from q or keep the token for ever.

Σ_e is called a *Producer*, if $\Sigma_\mathrm{e}^{q=0}$ eventually permanently marks the initially unmarked q. In case of a Producer environment, it is enough to place a token on q, as stuttering-invariant properties do not count the number of steps to generate the token on q.

We apply a *Dead End* reduction, if the place q is never marked in Σ. In case of a Dead End environment we can omit Σ_e and also the transitions of Σ_k that are connected to it. Transitions in ${}^\bullet q$ are never fired because otherwise q would be marked and since q is never marked, transitions in $q^\bullet$ are never enabled.

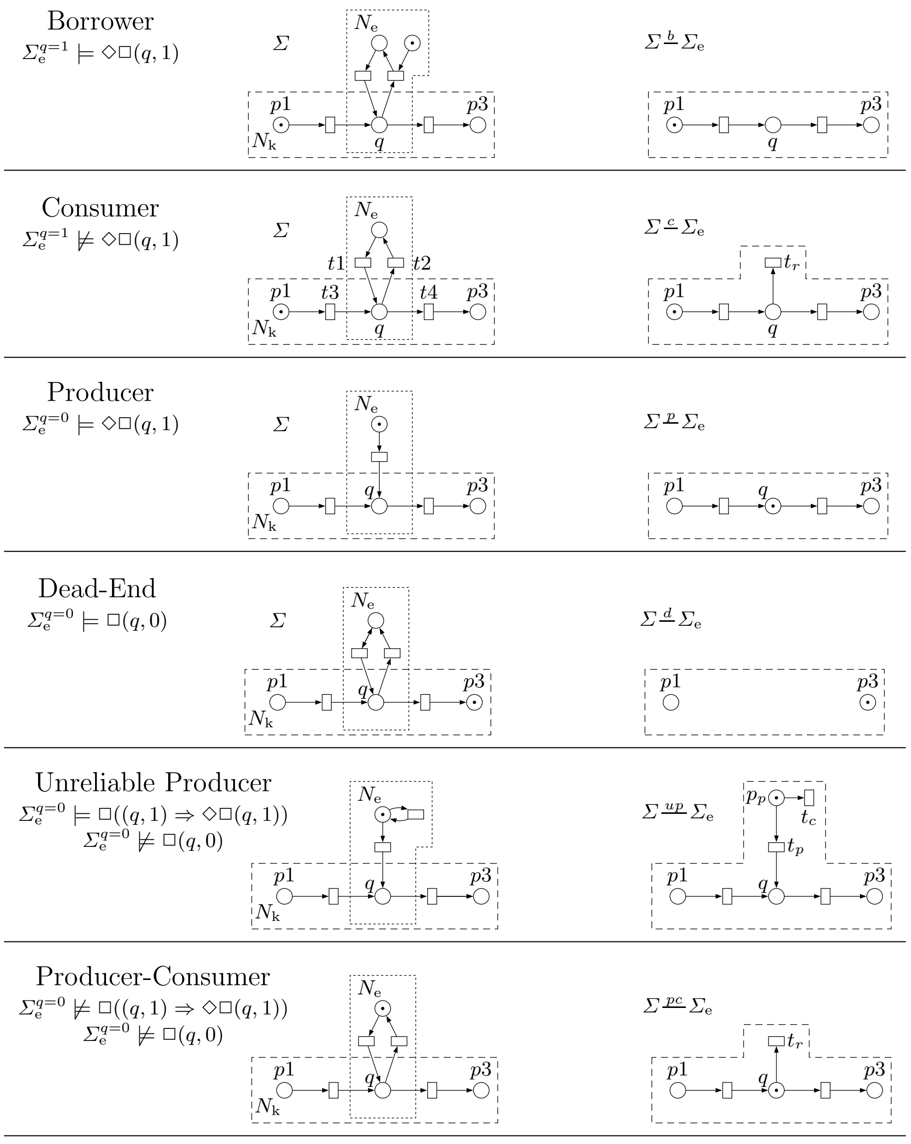

Borrower
$\Sigma_e^{q=1} \models \Diamond\Box(q,1)$
Σ
N_e
$p1$
$p3$
N_k
q
$\Sigma \overset{b}{—} \Sigma_e$
Consumer
$\Sigma_e^{q=1} \not\models \Diamond\Box(q,1)$
$t1$
$t2$
$t3$
$t4$
$\Sigma \overset{c}{—} \Sigma_e$
t_r
Producer
$\Sigma_e^{q=0} \models \Diamond\Box(q,1)$
$\Sigma \overset{p}{—} \Sigma_e$
Dead-End
$\Sigma_e^{q=0} \models \Box(q,0)$
$\Sigma \overset{d}{—} \Sigma_e$
Unreliable Producer
$\Sigma_e^{q=0} \models \Box((q,1) \Rightarrow \Diamond\Box(q,1))$
$\Sigma_e^{q=0} \not\models \Box(q,0)$
$\Sigma \overset{up}{—} \Sigma_e$
p_p
t_c
t_p
Producer-Consumer
$\Sigma_e^{q=0} \not\models \Box((q,1) \Rightarrow \Diamond\Box(q,1))$
$\Sigma_e^{q=0} \not\models \Box(q,0)$
$\Sigma \overset{pc}{—} \Sigma_e$

Fig. 1. The reductions

Σ_e is an *Unreliable Producer*, if $\Sigma_e^{q=0}$ eventually permanently marks q at some executions and never marks q at others. An Unreliable Producer subnet is replaced by a net that can do the same, i.e. produce a token or never mark q.

An environment subnet Σ_e is called a *Producer-Consumer*, if some executions of $\Sigma_e^{q=0}$ generate a token on q but do not eventually permanently mark q.

Definition 5 (Borrower, Consumer, Dead End, Producer, Unreliable Producer, Producer-Consumer)
Let Σ be reducible by an environment Σ_{e}. Let N_{k} be the kernel and q be the 1-safe contact place, $q \in (P_{\mathrm{k}} \cap P_{\mathrm{e}})$.

Σ_{e} *is a* Borrower
iff q is a 1-safe place of $\Sigma_{\mathrm{e}}^{q=1}$ and $\Sigma_{\mathrm{e}}^{q=1} \models \Diamond\Box(q,1)$.
The Borrower-reduced *of Σ by Σ_{e}, $\Sigma \xrightarrow{b} \Sigma_{\mathrm{e}}$, is the net $(P_{\mathrm{k}}, T_{\mathrm{k}}, W|_{(P_{\mathrm{k}},T_{\mathrm{k}})}, M_0|_{P_{\mathrm{k}}})$.*

Σ_{e} *is a* Consumer
iff q is a 1-safe place of $\Sigma_{\mathrm{e}}^{q=1}$ and $\Sigma_{\mathrm{e}}^{q=1} \not\models \Diamond\Box(q,1)$.
The Consumer-reduced *of Σ by Σ_{e}, $\Sigma \xrightarrow{c} \Sigma_{\mathrm{e}}$ is the net $\Sigma \xrightarrow{c} \Sigma_{\mathrm{e}} = (N', M_0|_{P_{\mathrm{k}}})$ with $N' = (P_{\mathrm{k}}, T_{\mathrm{k}} \uplus \{t_r\}, W|_{(P_{\mathrm{k}},T_{\mathrm{k}})} \uplus \{(q,t_r) \mapsto 1\})$.*

Σ_{e} *is a* Dead End
iff q is not 1-safe in $\Sigma_{\mathrm{e}}^{q=1}$ and $\Sigma_{\mathrm{e}}^{q=0} \models \Box(q,0)$.
The Dead End-reduced *of Σ by Σ_{e}, $\Sigma \xrightarrow{d} \Sigma_{\mathrm{e}}$, is $(P', T', W|_{(P',T')}, M_0|_{P'})$ with $P' = P_{\mathrm{k}} \setminus \{q\}$ and $T' = T_{\mathrm{k}} \setminus ({}^{\bullet}q \cup q^{\bullet})$.*

Σ_{e} *is a* Producer
iff $\Sigma_{\mathrm{e}}^{q=0} \models \Diamond\Box(q,1)$.
The Producer-reduced *of Σ by Σ_{e}, $\Sigma \xrightarrow{p} \Sigma_{\mathrm{e}}$, is $(P_{\mathrm{k}}, T_{\mathrm{k}}, W|_{(P_{\mathrm{k}},T_{\mathrm{k}})}, M_0^{q=1}|_{P_{\mathrm{k}}})$.*

Σ_{e} *is an* Unreliable Producer
iff $\Sigma_{\mathrm{e}}^{q=0} \not\models \Box(q,0)$ and $\Sigma_{\mathrm{e}}^{q=0} \models \Box((q,1) \Rightarrow \Diamond\Box(q,1))$.
The Unreliable Producer-reduced *of Σ by Σ_{e}, $\Sigma \xrightarrow{up} \Sigma_{\mathrm{e}}$, is the net $\Sigma \xrightarrow{up} \Sigma_{\mathrm{e}} = (P_{\mathrm{k}} \uplus \{p_p\}, T_{\mathrm{k}} \uplus \{t_c, t_p\}, W|_{(P_{\mathrm{k}},T_{\mathrm{k}})} \uplus \{(p_p,t_p) \mapsto 1, (t_p,q) \mapsto 1, (p_p,t_c) \mapsto 1\}, M_0|_{P_{\mathrm{k}}} \uplus \{p_p \mapsto 1\})$.*

Σ_{e} *is a* Producer-Consumer
iff $\Sigma_{\mathrm{e}}^{q=0} \not\models \Box(q,0)$ and $\Sigma_{\mathrm{e}}^{q=0} \not\models \Box((q,1) \Rightarrow \Diamond\Box(q,1))$.
The Producer-Consumer-reduced *of Σ by Σ_{e}, $\Sigma \xrightarrow{pc} \Sigma_{\mathrm{e}}$ is the net $\Sigma \xrightarrow{pc} \Sigma_{\mathrm{e}} = (P_{\mathrm{k}}, T_{\mathrm{k}} \uplus \{t_r\}, W|_{(P_{\mathrm{k}},T_{\mathrm{k}})} \uplus \{(q,t_r) \mapsto 1\}, M_0^{q=1}|_{P_{\mathrm{k}}})$.*

Note, that the contact place q is 1-safe in $\Sigma_{\mathrm{e}}^{q=0}$ in all cases. The place q is not 1-safe in $\Sigma_{\mathrm{e}}^{q=1}$ for a Dead End environment and the producing environment nets, that is $\Sigma \xrightarrow{p} \Sigma_{\mathrm{e}}$, $\Sigma \xrightarrow{up} \Sigma_{\mathrm{e}}$ and $\Sigma \xrightarrow{pc} \Sigma_{\mathrm{e}}$. But $\Sigma_{\mathrm{e}}^{q=1}$ does not need to be examined for the producing environments. All reductions guarantee that q remains 1-safe in the reduced net.

Each of these reduction rules preserves LTL-X, i.e., Σ satisfies an LTL-X property ϕ fairly w.r.t. T_{k} if and only if its reduced net Σ' satisfies ϕ.

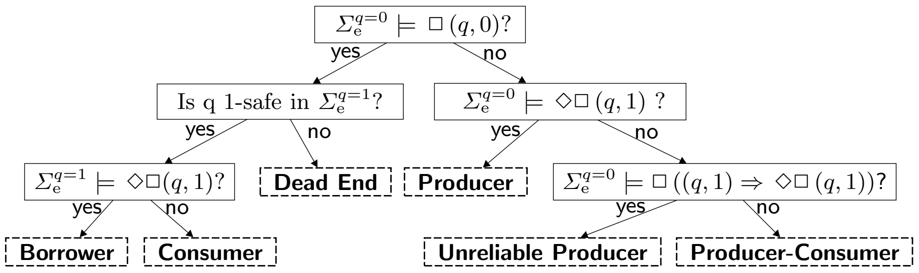
Theorem 6. *Let ϕ be an LTL-X formula. Let Σ be reducible by Σ_{e} according to Definition* 5 *and let Σ' be the corresponding reduced net. Let T_{k} be the transition set of Σ_{k}.*

$\Sigma \models \phi$ *fairly w.r.t* $T_{\mathrm{k}} \Leftrightarrow \Sigma' \models \phi$.

For some reduction rules even stronger results hold. The proof of these results can be found in the full version of this paper [11]. We briefly sketch the general structure of the proof. We show that any firing sequence of Σ that is fair w.r.t. T_{k} corresponds to a maximal firing sequence of Σ'. Also, any maximal firing sequence of Σ' corresponds to a firing sequence of Σ that is fair w.r.t. T_{k}. Corresponding firing sequences generate corresponding markings. Firing sequences σ of Σ and σ' of Σ' *correspond* iff they fire the same transitions of T_{k} in the same order. Markings M and M' correspond iff they coincide on places in $P_{\mathrm{k}} \setminus \{q\}$. Marking sequences that are generated by corresponding firing sequences satisfy the same LTL-$_{\mathrm{X}}$ formulas.

Figure 2 illustrates how the appropriate reduction rule to replace an environment net Σ_{e} can be determined. The decision tree also illustrates that the set of reduction rules is complete, i.e. every environment net is reducible by one of the reduction rules.

The reductions guarantee that the behaviour of the kernel subnet Σ_{k} is preserved and thus also its state space. Only for the Dead End reduction, Σ_{e} – more precisely $\Sigma_{\mathrm{e}}^{q=1}$ – may possibly expose spurious behaviour. $\Sigma_{\mathrm{e}}^{q=1}$ may even be unbounded, whereas Σ_{e} within Σ is bounded. During the evaluation of this method we never encountered such a case. For a 1-safe Petri net, Σ_{e} is already proved a Dead End, if $\Sigma_{\mathrm{e}}^{q=0}$ satisfies $\Box(q,0)$ and some place in $\Sigma_{\mathrm{e}}^{q=1}$ gets more than one token. Also, if earlier simulation showed that q can be marked, Σ_{e} is consequently not a Dead End.



$\Sigma_{\mathrm{e}}^{q=0} \models \Box(q,0)$?
yes
no
Is q 1-safe in $\Sigma_{\mathrm{e}}^{q=1}$?
$\Sigma_{\mathrm{e}}^{q=0} \models \Diamond\Box(q,1)$?
yes
no
yes
no
$\Sigma_{\mathrm{e}}^{q=1} \models \Diamond\Box(q,1)$?
Dead End
Producer
$\Sigma_{\mathrm{e}}^{q=0} \models \Box((q,1) \Rightarrow \Diamond\Box(q,1))$?
yes
no
yes
no
Borrower
Consumer
Unreliable Producer
Producer-Consumer

Fig. 2. Leafs of the decision tree classify Σ_{e}

In this section we have shown how to reduce a net Σ composed of a kernel Σ_{k} and an environment Σ_{e}, such that Σ_{k} and Σ_{e} only share a 1-safe place. First, LTL-$_{\mathrm{X}}$ formulas are checked on Σ_{e} to determine the appropriate reduction rule. The reduction rule replaces Σ_{e} by the corresponding summary net $S(\Sigma_{\mathrm{e}})$. In the next section we will present an algorithm that decomposes a given 1-safe net into a kernel and environment nets. Note, that in case there are several environment nets $\Sigma_{\mathrm{e}_1}, ..., \Sigma_{\mathrm{e}_n}$, the results of this section justify to check all nets simultaneously, since which reduction rule is applicable depends on Σ_{e_i} only.

4 The Decomposition Algorithm

In this section we present an algorithm that determines a decomposition of a given connected and 1-safe net Σ into (i) a kernel net Σ_{k} that contains all places mentioned by the temporal logic property ϕ and (ii) environment nets $\Sigma_{e_1}, ..., \Sigma_{e_n}$. The connectedness assumption simplifies the following but does not impose a strong restriction, as we can apply the linear time slicing algorithm of [12] and then consider every connected subnet of Σ on its own.

With a simple modification of Σ, the task of finding environment nets of Σ is basically the task of finding places $q \in (P \setminus \mathit{scope}(\phi))$ such that after removing q from Σ, Σ is not connected anymore. In terms of graph theory, this is the problem of finding articulation points $q \in (P \setminus \mathit{scope}(\phi))$. The set of articulation points can be determined in linear time by an depth-first search (DFS) algorithm. A good presentation of a DFS algorithm to determine articulation points can be found in [13].

Before we sketch the decomposition algorithm, we briefly introduce the terms *connected component* and *biconnected component* [14]. A subgraph is connected if there is a path between any two nodes. A connected component is a maximal connected subgraph. A subgraph is biconnected if any two nodes can be joined by two independent paths. A biconnected component is a maximal biconnected subgraph.

The decomposition algorithm can be sketched as follows:
Given a net Σ and a set of places $\mathit{scope}(\phi)$, we consider Σ as an undirected graph with nodes $P \cup T$ and edges $\{(n, n') \mid W(n, n') \neq 0\}$.

1. Generate the modified Σ, $\hat{\Sigma}$.
   Introduce a new transition t' and connect every place in $\mathit{scope}(\phi)$ to t', that is $\hat{\Sigma} := (P, T \uplus \{t'\}, W \cup \{(t', p) \mid p \in \mathit{scope}(\phi)\}, M_0)$. All places $p, p' \in \mathit{scope}(\phi)$ are biconnected in $\hat{\Sigma}$. One path between p and p' exists since Σ is connected and there is a second path via t'.
2. Perform a DFS to determine biconnected components of $\hat{\Sigma}$.
   Ignore articulation points $a \in (T \cup \mathit{scope}(\phi))$. The DFS uses a stack to keep track of the currently traversed component. Visited edges are put onto the stack. If a component has been found, all its edges are removed from the stack. The search starts at a place $p \in \mathit{scope}(\phi)$, so that the first and thus the last component on the stack is the biconnected component containing all places in $\mathit{scope}(\phi)$, that is the kernel $\hat{\Sigma}_k$.
3. Subtract the edges of $\hat{\Sigma}_k$ from $\hat{\Sigma}$.
4. Subtract unconnected places and transitions of $\hat{\Sigma}_k$ from $\hat{\Sigma}$.
   The remainder of $\hat{\Sigma}$ equals the reducible part of Σ.

There are several possibilities to continue. The theoretically best way to avoid the combinatorial blow-up of the state space is to repeatedly replace the smallest environments. For technical reasons, we implemented step 5 in our prototype as:

5. Perform a DFS to determine the connected components.
   Every connected component is an environment net and can thus be replaced.

This way we chose the greatest environment subnets.

Figure 3 illustrates the algorithm on an example. The two grey places are in $\mathit{scope}(\phi)$. Σ is already extended by t' to connect the places in $\mathit{scope}(\phi)$ (step 1).

The original net Σ consists of five biconnected components whereas the extended net $\hat{\Sigma}$ consists of three biconnected components only. The three biconnected components of Σ with articulation points $\{p_1\}$, $\{p_1, p_2\}$, and $\{p_2, p_3\}$, respectively, build one biconnected component in $\hat{\Sigma}$, that is $\hat{\Sigma}_{\mathrm{k}}$. In Fig. 3 (a) the maximal environments have been determined, as implemented by step 5. In Fig. 3 (b) the minimal environment nets have been determined. For the latter, we first model check Σ_{e_1}, which is then replaced, generating a reduced environment Σ'_{e_2}. Then the reduced environment net Σ'_{e_2} is model checked and replaced, generating a reduced kernel.

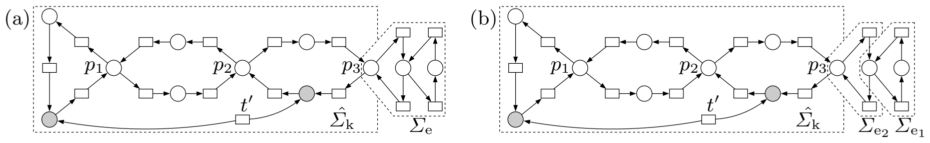


Fig. 3. Decomposition in kernel and (a) maximal and (b) minimal environments

In the next section we give experimental data gained with our prototype implementation of the presented algorithm. But let us first consider the complexity of the presented algorithm. Step 1 can be done in $O(|scope(\phi)|)$ time. To determine biconnected components of $\hat{\Sigma}$ via DFS (step 2), takes $O(|P|+|T|+1+|W|+|scope(\phi)|)$, where $|W| = |\{(x,y) \in ((P \times T) \cup (T \times P)) \mid W(x,y) \neq 0\}|$. Note that $\hat{\Sigma}$ has $|scope(\phi)|$ additional arcs and one additional transition. Steps 3+4 take $O(|P|+|T|+1+|scope(\phi)|)$. To perform a DFS to determine the connected components takes again $O(|P|+|T|+1+|W|+|scope(\phi)|)$. Hence the algorithm determines the greatest environment nets in $O(|P|+|T|+|W|+|scope(\phi)|)$ time.

The decomposition algorithm presented in this section can also be applied to non-1-safe nets, but to apply the reductions it has additionally to be guaranteed that the articulation points are 1-safe in Σ. To determine whether a given P/T net is 1-safe, is known to be PSPACE complete [15], but it may for instance be possible to determine a structural bound by linear programming techniques, which can be done in polynomial time [16].

5 Evaluation and Extensions

To evaluate our approach we used a set of widely used case studies[1]. All nets of the benchmark are known to be 1-safe. Most of them are taken from the benchmark set of Corbett [17]. The nets bruijn_2, rw_1w3r and dijkstra_2 are part of a benchmark in [19]. For all other examples we used the formulas of [20].

Table 1 sketches the benchmark. It displays the net size (number of places, transitions, arcs) of the original and the reduced net, the LTL$_{\text{-X}}$ property to check and the number of articulation points used of the total number of articulation points. Since the prototype determines maximal environment nets, some articulation points are not used for the decomposition.

[1] The benchmark is downloadable at [18].

Table 1. The benchmark nets

	original	reduced	property	$\frac{\#(\text{used articul.})}{\#(\text{articul.})}$
peterson	27,31,120	22,26,110	$\Box(\Diamond \neg P9)$	5/5
bds_1	82,63,342	5,5,17	$\Box(P83 \Rightarrow (\Diamond P82))$	1/6
rw_12	115,317,1890	5,5,17	$\Box(P115 \Rightarrow (\Diamond P114))$	1/2
dijkstra_2	68,86,324	62,80,312	$\Box(\neg P22 \,\|\|\, \neg P43)$	6/6
bruijn_2	86,165,777	79,158,763	$\Box(\neg P33 \,\|\|\, \neg P66)$	7/7
gas_nq_4	312,469,2770	311,468,2768	$\Diamond(\Box(P311 \,\|\|\, P312))$	1/1
over_5	145,99,546	5,5,17	$\Diamond(\Box(P144 \,\|\|\, P145))$	1/6
rw_1w3r	106,270,1172	96,260,1152	$\Box(P1 \Rightarrow (\Diamond P2))$	10/10

In Table 2 the number of states and state transitions is displayed that have been encountered while model checking the given LTL$_{-X}$ property using the Prod tool [21]. Column *original* displays the number of states and state transitions encountered while model checking the full net, *reduced* displays the results encountered while model checking the reduced net. The overhead for checking the LTL$_{-X}$ formulas to characterise the influence of an environment net is captured in column *interm.*. This column gives the sum of states and state transitions encountered at model checking the characterising formulas on environment nets. Column *saved* gives savings as percentage of the original (cf. first column).

Table 2. State space savings by the reductions

	original	reduced	interm.	saved [%]
peterson	186,617	76,199	40,40	37.6,61.2
bds_1	72271,790174	8,11	36189,263406	49.9,66.6
rw_12	8222,147536	8,11	4199,49276	48.8,66.6
dijkstra_2	2725,9243	492,995	48,48	80.1,88.7
bruijn_2	5184,19112	1097,2383	56,56	77.7,87.2
gas_nq_4	18385,59906	18250,59393	8,8	0.6,0.6
over_5	45095,221113	5,5	33563,163675	25.5,25.9
rw_1w3r	330545,1573296	44569,141318	80,80	86.4,91.0

Similar to Table 2, Table 3 displays the results of model checking the given properties, but this time Prod uses partial order reduction [9]. The columns *interm.* and *saved* display again the number of states and state transitions encountered while model checking the characterising formulas and the number of saved states and state transitions, respectively. For the nets peterson, dijkstra_2, bruijn_2 and rw_1w3r our approach performs worse than using partial order reduction only. Analysis showed that the biconnected components were very small and thus the gain of applying our reductions was not sufficient. This inspired to optimise our algorithm by *micro reduction rules*. The appropriate reduction rule for an environment consisting of an articulation point, a transition and up to one additional place is determined by matching the net structure instead of model

Table 3. Reduction results with partial order reduction

	original	reduced	interm.	saved [%]	interm. (micro)	saved [%] (micro)
peterson	65,139	58,129	40,40	-50.7,-50.7	0,0	10.7,7.1
bds_1	1498,3610	8,11	788,1247	46.8,65.1	788,1247	46.8,65.1
rw_12	8222,147536	8,11	4199,49276	48.8,66.5	4199,49276	48.8,66.5
dijkstra_2	422,645	416,639	48,48	-9.9,6.5	0,0	1.4,0.9
bruijn_2	766,1192	759,1185	56,56	-6.3,-4.1	0,0	0.9,0.5
gas_nq_4	12561,31038	12560,31031	8,8	∼ 0,0	0,0	∼0,0
over_5	7379,12861	5,5	6102,10704	17.2,16.7	6102,10704	17.2,16.7
rw_1w3r	15637,26452	15617,26432	80,80	-0.3,-0,2	0,0	∼0.1,0

checking. The result of applying the modified reduction algorithm using these micro reductions is displayed at columns *interm. (micro)* and *saved (micro)*. The results show that our approach can help to tackle the state explosion problem. The approach is most efficient for nets with big environment nets like bds_1, rw_12 and over_5. For these nets even the combination without micro reductions has a beneficial effect. The nets peterson, dijkstra_2, bruijn_2, gas_nq_4 and rw_1w3r have micro environments only. For these nets our approach causes an overhead but with micro reductions it is beneficial for all considered examples.

6 Discussion and Conclusion

Effectiveness. We see the main contributions of this paper to be a decomposition algorithm that partitions the net efficiently and generates components so that spurious behaviour is unlikely. The approach speeds up model checking as the combinatorial blow-up caused by concurrent behaviour of kernel and environments is avoided by reducing the environments. For 1-safe nets the decomposition can be determined in linear time. Spurious behaviour is possible only in case Σ_e is a Dead End. Given we know from previous analysis that the contact place is eventually marked, there is no risk to encounter spurious behaviour.

To determine the appropriate reduction the environment net is model checked three times (cf. Fig. 2). Thus our method may examine a slightly greater number of states than by model checking the original system and only in case there is very little concurrency within kernel and environment net. Micro reductions are a measure to deal with the smallest environment nets that expose no concurrency.

Restriction to 1-safe Nets. The correctness results of Sect. 3 apply not only to 1-safe nets. Also the algorithm presented in Sect. 4 can decompose any P/T net. But to apply the reduction it has to be guaranteed that the articulation points are 1-safe (for instance by a structural bound). We plan to examine the effect of our approach on bounded Petri nets, extending the decomposition algorithm to check whether an articulation point is structurally 1-bounded.

Decomposition Approaches. The results in Sect. 5 were gained with our first prototype that replaces maximal environment nets. Recent experiments with a second implementation have shown that replacing minimal environments performs better when the full state space is considered, but an algorithm replacing maximal environments performs better when partial order reductions are applied. We are currently examining this unexpected experimental effect.

Conclusion. We presented a decompositional approach to alleviate the state explosion problem of model checking an LTL$_{\text{-X}}$ formula ϕ. We suggested to decompose the Petri net into a kernel net containing $scope(\phi)$ and environment nets so that articulation points of the Petri net graph are interfaces between kernel and environment nets. For a 1-safe net the decomposition can be determined in linear time and every environment net can be replaced. To determine the applicable reduction an environment net is checked in isolation, thus avoiding the combinatorial blow-up. The application of our prototype to several case studies has shown good results. The evaluation also indicates that our approach can help to further accelerate model checking when combined with partial order reductions.

Acknowledgements. I thank Jan Rakow and Eike Best for reading and commenting this paper.

References

1. Lee, W.J., Cha, S.D., Kwon, Y.R., Kim, H.N.: A Slicing-based Approach to Enhance Petri Net Reachability Analysis. Journal of Research and Practice in Information Technology 3, 131–143 (2000)
2. Klai, K., Petrucci, L., Reniers, M.: An Incremental and Modular Technique for Checking LTL\X Properties of Petri nets. In: Derrick, J., Vain, J. (eds.) FORTE 2007. LNCS, vol. 4574, pp. 280–295. Springer, Heidelberg (2007)
3. Valmari, A.: Compositional Analysis with Place-Bordered Subnets. In: Valette, R. (ed.) ICATPN 1994. LNCS, vol. 815, pp. 531–547. Springer, Heidelberg (1994)
4. Berthelot, G.: Checking Properties of Nets Using Transformation. In: Rozenberg, G. (ed.) APN 1985. LNCS, vol. 222, pp. 19–40. Springer, Heidelberg (1986)
5. Desel, J., Esparza, J.: Free choice Petri Nets. Cambridge University Press, New York (1995)
6. Poitrenaud, D., Pradat-Peyre, J.-F.: Pre- and Post-agglomerations for LTL Model Checking. In: Nielsen, M., Simpson, D. (eds.) ICATPN 2000. LNCS, vol. 1825, pp. 387–408. Springer, Heidelberg (2000)
7. Esparza, J., Schröter, C.: Net Reductions for LTL Model-Checking. In: Margaria, T., Melham, T.F. (eds.) CHARME 2001. LNCS, vol. 2144, pp. 310–324. Springer, Heidelberg (2001)
8. Haddad, S., Pradat-Peyre, J.-F.: New Efficient Petri Nets Reductions for Parallel Programs Verification. In: Parallel Processing Letters, vol. 16(1), pp. 101–116. World Scientific Publishing Company, Singapore (2006)
9. Valmari, A.: On-the-Fly Verification with Stubborn Sets. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 397–408. Springer, Heidelberg (1993)
10. Lamport, L.: What Good is Temporal Logic? In: Information Processing 1983: Proceedings of the IFIO 9th World Computer Congress, pp. 657–668 (1983)

11. Rakow, A.: Decompositional Petri Net Reductions. Technical Report (June 2008), http://parsys.informatik.uni-oldenburg.de/~astrid3/ifm/reducts.pdf
12. Rakow, A.: Slicing Petri nets with an Application to Workflow Verification. In: Geffert, V., Karhumäki, J., Bertoni, A., Preneel, B., Návrat, P., Bieliková, M. (eds.) SOFSEM 2008. LNCS, vol. 4910, pp. 436–447. Springer, Heidelberg (2008)
13. http://sparcs.kaist.ac.kr/~lacrimosa/algorithm/2003/CS300-09.ppt
14. Diestel, R.: Graph Theory. Graduate Texts in Mathematics, vol. 173. Springer, Heidelberg (2005)
15. Cheng, A., Esparza, J., Palsberg, J.: Complexity Results for 1-safe nets. In: Shyamasundar, R.K. (ed.) FSTTCS 1993. LNCS, vol. 761, pp. 326–337. Springer, Heidelberg (1993)
16. Girault, C., Valk, R.: Petri Nets for System Engineering: A Guide to Modeling, Verification, and Applications. Springer, New York (2001)
17. Corbett, J.C.: Evaluating Deadlock Detection Methods for Concurrent Software. IEEE Transactions on Software Engineering 22(3), 161–180 (1996)
18. http://parsys.informatik.uni-oldenburg.de/~astrid3/ifm/bm.tar.gz
19. Esparza, J., Heljanko, K.: Implementing LTL Model Checking with Net Unfoldings. Research Report A68, Laboratory for Theoretical Computer Science, Helsinki University of Technology, Espoo, Finland, 29p. (March 2001)
20. Schröter, C., Khomenko, V.: Parallel LTL-X Model Checking of High-Level Petri Nets Based on Unfoldings. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 109–121. Springer, Heidelberg (2004)
21. http://www.tcs.hut.fi/Software/prod/

Author Index

Abbasi, Naeem 277
Abrial, Jean-Raymond 1
Al-Rawi, Bashar 322

Basin, David 1
Beneš, Nikola 307
Boiten, Eerke 183
Brim, Lubos 307
Bui, Thang H. 72
Bultan, Tevfik 167
Butler, Michael 20

Cabot, Jordi 40
Cavalcanti, Ana 151
Černá, Ivana 307
Clarisó, Robert 40
Colvin, Robert 118
Cook, Byron 39

Derrick, John 183
Dovland, Johan 215

Echtle, Klaus 200

Fraikin, Benoît 337
Frappier, Marc 337

Garavel, Hubert 88
Garis, Ana G. 231
George, Chris 231
Gibbons, Jeremy 56
Grosu, Radu 322

Hasan, Osman 277
Hayes, Ian J. 118
Hoang, Thai Son 1
Huang, Xiaowan 322

Johnsen, Einar Broch 215

Kuruma, Hironobu 1

Lämmel, Ralf 246
Lang, Frédéric 88
Langari, Zarrin 261

Milhau, Jérémy 337

Nymeyer, Albert 72

Owe, Olaf 215

Pace, Gordon J. 292
Parisaca Vargas, Abigail 231

Rakow, Astrid 352
Riera, Daniel 40

Sakallah, Karem 322
Salaün, Gwen 167
Schneider, Gerardo 292
Schneider, Steve 103
Smolka, Scott 322
Sochor, Jiri 307
Sorea, Maria 200
Steffen, Martin 215
Stöcker, Jan 88

Tahar, Sofiène 277
Tarifa, S. Lizeth Tapia 231
Trefler, Richard 261
Treharne, Helen 103

Vařeková, Pavlina 307
Voss, Sebastian 200

Weiß, Benjamin 136
Wong, Peter Y.H. 56

Yang, Zijiang 322

Zaytsev, Vadim 246
Zeyda, Frank 151
Zimmerova, Barbora 307