Volume Editors

Maurice H. ter Beek  
Istituto di Scienza e Tecnologie dell’Informazione (ISTI-CNR)  
Area della Ricerca di Pisa, Via G. Moruzzi 1, 56124 Pisa, Italy  
maurice.terbeek@isti.cnr.it

Niels Lohmann  
Universität Rostock, Institut für Informatik  
18051 Rostock, Germany  
niels.lohmann@informatik.uni-rostock.de
Preface

Web services are fundamental to cloud computing and other computing paradigms based on service-oriented architectures and applications. They make functional and autonomous building blocks available over the Internet, independent of platforms and programming languages, and both within and across organizational boundaries. These can then be described, located, orchestrated, and invoked. Virtualization technology has moreover led to the Software as a Service, Platform as a Service, and Infrastructure as a Service notions.

Formal methods can play a fundamental role in research on these concepts. They can help define unambiguous semantics for the languages and protocols that underpin web service infrastructures, and provide a basis for checking the conformance and compliance of bundled services. They can also empower dynamic discovery and binding with compatibility checks against behavioral properties, quality of service requirements, and service-level agreements. The resulting possibility of formal verification and analysis of (security) properties and performance (dependability and trustworthiness) is essential to cloud computing and to application areas like e-commerce, e-government, e-health, workflow, business process management, etc. Moreover, the challenges raised by research on these concepts can extend the state of the art in formal methods.

The aim of the WS-FM workshop series is to bring together researchers working on Web Services and Formal Methods in order to catalyze fruitful collaboration. Its scope is not limited to technological aspects. In fact, there is a strong tradition of attracting submissions on formal approaches to enterprise systems modeling in general, and business process modeling in particular. Potentially, this may have a significant and lasting impact on the ongoing standardization efforts in cloud computing technologies. Previous editions took place in Pisa (2004), Versailles (2005), Vienna (2006), Brisbane (2007), Milan (2008), Bologna (2009), Hoboken (2010), and Clermont-Ferrand (2011).

Following the success of the previous workshops, the 9th International Workshop on Web Services and Formal Methods (WS-FM 2012) took place on 6 and 7 September 2012 in Tallinn, Estonia, co-located with the 10th International Conference on Business Process Management (BPM 2012).

The contributions in this volume cover aspects such as the modeling and analysis of web services, service discovery, and service coordination, with formal methods like BPEL, CSP, Maude, and Petri nets.

The workshop program includes keynotes by Farouk Toumani from the Blaise Pascale University Aubière and Emilio Tuosto from the University of Leicester, and papers from researchers across the globe — including Canada, China, Estonia, Germany, Italy, The Netherlands, and Portugal. The workshop initially received a total of 19 submissions, which were each reviewed by at least 3 researchers from a strong program committee of international reputation. After lively discussions,
the committee eventually decided to accept 8 papers. These proceedings include all accepted submissions, appropriately updated in light of the reviews.

We wish to thank the program committee and the external reviewers for their timely reviewing. We acknowledge the unbeatable support of EasyChair for managing the reviewing process. Finally, we wish to thank Marlon Dumas for his excellent organization of both BPM and WS-FM.

August 2012

Maurice ter Beek
Niels Lohmann
Organization

Program Committee Co-chairs

Maurice H. ter Beek       ISTI–CNR, Pisa, Italy
Niels Lohmann             Universität Rostock, Germany

Program Committee

Farhad Arbab             CWI, Amsterdam, The Netherlands
Laura Bocchi             University of Leicester, UK
Mario Bravetti           University of Bologna, Italy
Roberto Bruni            University of Pisa, Italy
Marco Carbone            IT University of Copenhagen, Denmark
Schahram Dustdar         Vienna University of Technology, Austria
José Luiz Fiadeiro       Royal Holloway, University of London, UK
Stefania Gnesi           ISTI–CNR, Pisa, Italy
Lars Grunske             University of Kaiserslautern, Germany
Sylvain Hallé            Université du Québec à Chicoutimi, Canada
Ivan Lanese              University of Bologna, Italy
Manuel Mazzara           Newcastle University, UK
Arjan Mooij               Embedded Systems Institute, The Netherlands
Jean-Marc Petit          University of Lyon/CNRS, France
Artem Polyvyanyy         HPI Potsdam, Germany
Rosario Pugliese         University of Florence, Italy
Christian Stahl          Eindhoven University of Technology, The Netherlands
Erik de Vink             Eindhoven University of Technology, The Netherlands
Hagen Voelzer            IBM Research, Switzerland
Matthias Weidlich        Technion, Israel
Martin Wirsing           Ludwig-Maximilians-Universität München, Germany
Karsten Wolf             Universität Rostock, Germany

Additional Reviewers

Alexei Iliasov           Sung-Shik T.Q. Jongmans
Fabrizio Montesi        Julien Lange
Giorgio Oronzo Spagnolo  Francesco Santini
Sara Fernandes
Steering Committee

<table>
<thead>
<tr>
<th>Name</th>
<th>Institution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wil M. P. van der Aalst</td>
<td>Eindhoven University of Technology, The Netherlands</td>
</tr>
<tr>
<td>Mario Bravetti</td>
<td>University of Bologna, Italy</td>
</tr>
<tr>
<td>Marlon Dumas</td>
<td>University of Tartu, Estonia</td>
</tr>
<tr>
<td>José Luiz Fiadeiro</td>
<td>Royal Holloway, University of London, UK</td>
</tr>
<tr>
<td>Gianluigi Zavattaro</td>
<td>University of Bologna, Italy</td>
</tr>
</tbody>
</table>
# Table of Contents

## Invited Talks

Formal approaches for automatic synthesis of web service business protocols  
*Farouk Toumani*  
1

“I have read and agree with the terms and conditions” . . . so what? –  
Taking contracts for services seriously  
*Emilio Tuosto*  
2

## Regular Papers

Service discovery with cost thresholds  
*Jan Sürmeli*  
3

Private View Conformance Checking  
*Richard Müller, Wil M. P. Van Der Aalst, and Christian Stahl*  
18

Event Structures as a Foundation for Process Model Differencing,  
Part 1: Acyclic Processes  
*Abel Armas-Cervantes, Luciano García-Bañuelos, and Marlon Dumas*  
33

Formal Modeling and Analysis of the REST Architecture Using CSP  
*Xī Wu, Yue Zhang, Huibiao Zhu, Yongxin Zhao, Zailiang Sun, and Peng Liu*  
49

SiteHopper: Abstracting Navigation State Machines for the Efficient Verification of Web Applications  
*Guillaume Demarty, Fabien Maronnaud, Gabriel Le Breton, and Sylvain Hallé*  
65

Preference and Similarity-based Behavioral Discovery of Services  
*Farhad Arbab and Francesco Santini*  
80

Reconfiguration mechanisms for service coordination  
*Nuno Oliveira and Luís S. Barbosa*  
96

Linking the Semantics of BPEL using Maude  
*Peng Liu and Huibiao Zhu*  
112
Formal approaches for automatic synthesis of web service business protocols

Farouk Toumani

Blaise Pascal University, Laboratoire LIMOS - CNRS Clermont-Ferrand, France

Abstract. One of the ultimate goals of the web service technology is to enable rapid low-cost development and easy composition of distributed applications, a goal that has a long history strewn with only partial successes. The research problems underlying service composition are varied in nature and depend on several parameters such as the model used to describe the services, the communication model or the composition language. A line of demarcation between existing works in this area lies in the nature of the composition process: manual v.s. automatic. The first category of work deals generally with high-level composition design and programming details related to implementation issues while automatic service composition focuses on different issues such as composition verification, planning or synthesis.
In this talk, we consider more particularly the composition synthesis problem, i.e., the automated construction of a new target service by reusing some existing ones. We will review recent research works and challenges related to automatic synthesis of service composition and discuss the associated computational problems both in bounded and unbounded settings.
“I have read and agree with the terms and conditions”...so what?
Taking contracts for services seriously

Emilio Tuosto
emilio@mcs.le.ac.uk
Department of Computer Science,
University of Leicester, UK

The customary practice to regulate the functioning of services is to set terms & conditions. These are legal documents constraining the way services should be used and (sometimes) specifying what providers offer. In such documents it is not rare to find statements like

“We do our best to keep Facebook safe, but we cannot guarantee it”

(notably, safety is not defined in [1]!)

I argue that —despite being a practical way out— this is far from being ideal. For instance, the lack of precise guarantees is a main deterrent for industries wishing to move their applications and business to the cloud. Quoting from [2],

“Absent radical improvements in security technology, we expect that users will use contracts and courts, rather than clever security engineering, to guard against provider malfeasance.”

The key point is that terms & conditions should not be left to lawyers and courts only; rather computer scientists and IT practitioners should strive for robust techniques and methodologies capable of specifying formal contracts amenable of verification.

To support my contention I will overview some research recently carried out to address those issues.

References

Service discovery with cost thresholds

Jan Sürmeli
Humboldt-Universität zu Berlin, Institut für Informatik,
Unter den Linden 6, D-10099 Berlin, Germany
suermeli@informatik.hu-berlin.de

Abstract There exist several approaches to analyze a behavioral model of a stateful service \( N \). Such techniques frequently consider the concept of a partner. Intuitively, service \( Q \) is a partner of \( N \), if \( N \) and \( Q \) properly interact. So far, existing techniques do not consider non-functional properties, such as cost thresholds. Likewise, existing techniques to analyze non-functional properties neglect the behavior.

In this paper, we introduce a compact formal framework to enhance behavioral models with costs. Thereby, we introduce the concept of a cost bounded partner. Intuitively, a partner \( Q \) of \( N \) is cost bounded, if costs stay below a given threshold. For a class of enhanced models, we present a transformation procedure from an enhanced model \( N \) to an equivalent behavioral model \( N' \). Thereby we guarantee that analysis results of \( N' \) are also valid for \( N \).

1 Introduction

In a service-oriented architecture [8] (SOA), processes are composed from services. We understand a service as a stateful component, interacting with other services by means of asynchronous message exchange. Service discovery is the task to find a matching partner service \( Q \) for a given service \( N \). The SOA-triangle describes an instance of this problem: A service requester \( N \) queries a service broker \( B \) for matching service provider \( Q \). Thereby, we assume that \( Q \) published a service description in a repository accessed by \( B \).

Service broker \( B \) ensures the compatibility of requester \( N \) and provider \( Q \). If \( N \) and \( Q \) are compatible, we call them partners. In our setting, we assume that service descriptions are formal models. Here, \( B \) applies analysis and verification techniques to check compatibility between \( N \) and a candidate \( Q \). If \( N \) and \( Q \) are indeed partners, \( B \) returns \( Q \). Otherwise, it continues the search for a partner. Compatibility, and thus the concept of partner, may be defined on different levels [7]. For example, two services may be partners from a behavioral point of view; that is, they interact and terminate properly. However, this does not imply compatibility on other levels, for example for non-functional properties like costs, and time.

Running example. Consider a requester \( N \), and three providers \( Q_1, \ldots, Q_3 \), depicted in Fig. 1. The four services are modeled as open nets, Petri nets extended
by interfaces and sets of final markings. Transitions are depicted as rectangles, whereas places are depicted as circles. A dot or number on a place represents a token or a respective number of tokens on this place in the initial marking. Places drawn on the dashed line indicate the interface for asynchronous message exchange. Intuitively, composition is realized by gluing the services at respective interface places. The sets of final markings are not directly depicted but may be found in the figure caption. We describe the behavior of each depicted open net. Requester $N$ initiates communication by firing transition $A$, thereby sending an ORDER to its partner. Afterwards, it waits for an incoming payment option: Either a message NOW, enabling transition $B$, or a message LATER, enabling transition $C$. Firing one of the transitions $B$ and $C$ results in the single final marking $[p]$ of $N$. In the final marking, transition $D$ is enabled. Firing $D$ resets $N$ into its initial state. Provider $Q_1$ initially waits for an ORDER to arrive, enabling transition $E$. Once $E$ fires, $Q_1$ nondeterministically chooses between sending NOW and sending LATER, firing transitions $F$ or $G$ respectively. The empty marking $[]$ is the single final marking of $Q_1$. Provider $Q_2$ has a similar structure. However, $Q_2$ does not make the choice between NOW and LATER. Instead, $Q_2$ always replies with LATER. Provider $Q_3$ shows more complex behavior than $Q_1$ and $Q_2$. Intuitively, $Q_3$ behaves differently depending on the “round”: In the first round, $Q_3$ replies with NOW. Starting with the second round, it always replies with LATER.

Studying our running example, we observe that $N$ properly interacts with each of the providers $Q_1, \ldots, Q_3$: From any reachable state, a common final state may be reached. We study an example for a nonfunctional property. Consider that paying immediately forces $N$ to use a payment service requiring a fixed fee for each transaction. In contrast to that, paying later is free. Then, we observe that interaction with (1) $Q_1$ may result in arbitrarily high costs, (2) $Q_2$ is free, and (3) $Q_3$ always results in the same amount, namely the fixed fee for one transaction. Set the fixed fee to 10 units per transaction. Assume that $N$ fixes an acceptable
We write \( \mathbb{Q} \) is unacceptable, \( \mathbb{Q} \) is acceptable, and \( \mathbb{Q} \) is acceptable as well.

**Problem and contributions.** There exist analysis and verification techniques in literature for behavioral compatibility criteria, for instance deadlock freedom, or weak termination. However, these techniques consider purely behavioral models, completely abstracting from non-functional properties.

In this paper, we present a formal framework to enhance the purely behavioral models with costs. Thereby, we introduce cost specifications, consisting of a cost function to annotate transitions with costs, a cost model to specify how costs are aggregated, and a cost threshold, defining an acceptable upper bound for costs. However, a framework to specify costs is not very useful without proper analysis techniques. Thus, we study how existing techniques may be reused for a restricted class of cost specifications. Intuitively, we combine a purely behavioral model \( N \) with a cost specification \( S \) in a new behavioral model \( N_S \). As a result, one may analyze \( N_S \) instead of \( N \).

**Outline.** We structure our paper as follows: We repeat mathematical preliminaries, and the notion of open nets to formally model services in Sect. 2. We study properties of Petri nets in Sect. 3. We introduce our formal framework for enhancing behavioral models with costs in Sect. 4. We elaborate our approach for the class of worst case total cost specifications in Sect. 5. We discuss related work in Sect. 6, and conclude our paper in Sect. 7.

## 2 Preliminaries

We write \( \mathbb{N}_0 \) for the set of all natural numbers, starting with 0. Occasionally, we extend the natural numbers by the symbol \( \infty \not\in \mathbb{N}_0 \), denoted by \( \mathbb{N}_{0,\infty} := \mathbb{N}_0 \cup \{\infty\} \). In that case, we extend the order \( \leq \) on \( \mathbb{N}_0 \) canonically: For all \( x \in \mathbb{N}_{0,\infty} \): \( x \leq \infty \).

We define the power set \( \mathcal{P}(A) \) of some set \( A \) by the set of all its subsets; that is, \( \mathcal{P}(A) := \{ B \mid B \subseteq A \} \).

Let \( A \) be an alphabet. We write \( A^* \) for the set of all finite sequences over \( A \). Let \( B \subseteq A \) and \( \sigma \in A^* \). We define the restriction \( \sigma|_B \) of \( \sigma \) to \( B \) recursively:

\[
\epsilon|_B = \epsilon, \text{ for } b \in B, \quad (\sigma b)|_B := \sigma|_B b, \text{ and for } a \in A \setminus B, \quad (\sigma a)|_B := \sigma|_B.
\]

Let \( R \) be a binary relation. We define \( R^{-1} := \{ [b, a] \mid [a, b] \in R \} \). We define \( R(a) := \{ b \mid [a, b] \in R \} \) for \( a \in A \).

A bag (or: multiset) generalizes a set by assigning a multiplicity to each element. Formally, a function \( m : A \to \mathbb{N}_0 \) from some set \( A \) into \( \mathbb{N}_0 \) is a bag over \( A \). We write \( \text{Bags}(A) \) for the set of all bags over \( A \). If \( k \in \mathbb{N}_0 \) and \( m \in \text{Bags}(A) \), we write \( m_{>k} \) for the set of all elements which occur at least \( k + 1 \) times. That is, \( m_{>k} := \{ a \mid a \in A, m(a) > k \} \). Given a set \( B \), we define the canonical \( B \)-transformation \( m_B \) of \( m \) as the bag mapping each element \( a \) from the intersection of \( A \) and \( B \) to \( m(a) \), and all other arguments to zero. Formally, \( m_B \in \text{Bags}(B) \) is defined by \( x \mapsto m(x) \) if \( x \in A \), and \( x \mapsto 0 \), otherwise. We omit the index \( B \) if it is clear from the context. Given a bag \( m' \in \text{Bags}(B) \), we define \( m + m' \)
by component-wise addition, treating missing values as zero. Formally, the bag \((m + m') \in \text{Bags}(A \cup B)\) is defined by \(x \mapsto m_B(x) + m'_B(x)\). Bag \(m \in \text{Bags}(A)\) covers bag \(m' \in \text{Bags}(A)\), written \(m \succeq m'\) iff for all \(a \in A\), \(m(a) \geq m'(a)\). If \(m \succeq m'\), we define \((m - m') \in \text{Bags}(B)\) by \(a \mapsto m(a) - m'(a)\). For two sets \(F, G \subseteq \text{Bags}(A)\) of bags over \(A\), we define \((F + G) := \{(m + m') | m \in F, m' \in G\}\).

Let \(\mathcal{A}\) be a set, and \(\leq\) be a partial order on \(\mathcal{A}\). Let \(\mathcal{A} \subseteq \mathcal{A}\) and \(b \in \mathcal{A}\). We write \(b \preceq A\) iff (1) for all \(a \in A\), it holds that \(a \preceq b\), and (2) there exists \(a \in A\) with \(b \preceq A\). Similarly, we write \(b \succeq A\) iff (1) for all \(a \in A\), it holds that \(a \succeq b\), and (2) there exists \(a \in A\) with \(b \succeq A\).

### 2.1 Petri nets and Open nets

We model services as open nets \([5]\), a special class of Petri nets \([9]\). We quickly recall the syntax and semantics of Petri nets. Let \(P\) and \(T\) be finite, disjoint sets, and \(V: (P \times T) \cup (T \times P) \to \mathbb{N}_0\). Then, \(N = [P,T,V]\) is a Petri net structure. We call \(P\) the places, and \(T\) the transitions of \(N\). We call \(V\) the flow function of \(N\). A place models a store, buffer, or condition, a transition represents an action. A Petri net has a visual representation, where places are drawn as circles, transitions are drawn as rectangles, and \(V\) is realized by inscribed arcs: If \([x,y] \in (P \times T) \cup (T \times P)\), and \(V(x,y) > 0\), then we draw an arc inscribed with \(V(x,y)\) from \(x\) to \(y\). If \(V(x,y) = 1\), we usually omit the inscription.

Places of a Petri net structure can be marked by tokens. A token represents an arbitrary object, resource, or value. A function assigning a natural number of tokens to each place \(p \in P\) is a marking of \(P\). Hence, a marking \(m\) is a bag over \(P\), and we frequently use the bag notation. Transitions consume and produce markings as given by the flow function \(V\), thus changing the marking of the Petri net. A transition \(t \in T\) is enabled in a marking \(m\) in Petri net structure \(N\) iff for each place \([p,t] \in P \times \{t\}\), we find \(m(p) \geq V(p,t)\). Hence, a transition is enabled iff the current marking constitutes the necessary number of tokens on each place connected with \(t\). An enabled transition may fire, resulting in a new marking: If \(t\) is enabled in \(m\) in \(N\), then the resulting marking is \(m'\) defined by \(p \mapsto m(p) - V(p,t) + V(t,p)\). We call the triple \([m,t,m']\) a step of \(N\), often written as \(m \xrightarrow{t} N m'\). Consecutive steps \(m_1 \xrightarrow{t_1} N m_2 \xrightarrow{t_2} \cdots \xrightarrow{t_n} N m_{n+1}\) are frequently written in the abbreviated form \(m_1 \xrightarrow{t_1, \ldots, t_n} N m_{n+1}\). We further define \(m \xrightarrow{\epsilon} m'\) for all markings \(m\) of \(N\).

A Petri net \(N = [P,T,V,m^0]\) consists of a Petri net structure \([P,T,V]\), and a marking \(m^0\) of \([P,T,V]\), called initial marking of \(N\). Consecutive steps starting in \(m^0\) induce a run of \(N\): An executable transition sequence of \(N\). We define \(\text{Runs}(N) = \{\sigma | m^0 \xrightarrow{\sigma} N m\}\).

An open net is a Petri net with an interface, and a set of final markings. The interface consists of input places, and output places, modeling receiving and sending of messages, respectively. A final marking represents a state where termination is acceptable.

**Definition 1 (Open net).** An open net \(N = [P,T,V,m^0,M^I,I,O]\) extends a Petri net \([P,T,V,m^0]\) by a set of final markings \(M^I\), a set of input places \(I \subseteq P\),
As for any system, we may formulate properties for Petri nets. The following

**Definition 5 (Valid abstraction of run predicates).** A state predicate \( \phi \) of

N is a valid abstraction of \( \psi \) w.r.t. N, iff for all \( \sigma \) with \( m^0 \xrightarrow{\sigma} N m \), it holds

\( \psi(\sigma) \Leftrightarrow \phi(m) \).
A valid abstraction of $\psi$ w.r.t. $N$ does not necessarily exist: Consider two runs $\sigma, \sigma'$ yielding the same result marking $m$, and $\psi(\sigma) \neq \psi(\sigma')$. Then, it is impossible to find a valid abstraction. However, for a given run predicate $\psi$, there exists a class of Petri nets, such that one can find a valid abstraction. We call such a Petri net \textit{conclusive} w.r.t. $\psi$.

Formally, we first define \textit{conclusiveness} on markings, and extend this definition to Petri nets. A marking $m$ is conclusive w.r.t. $\psi$ iff for any two runs $\sigma, \sigma'$ resulting in $m$, we find $\psi(\sigma) \Leftrightarrow \psi(\sigma')$. A Petri net is conclusive iff all reachable markings are conclusive.

\textbf{Definition 6 (Conclusiveness).} A marking $m$ of $N$ is conclusive w.r.t. $\psi$, iff for all $\sigma, \sigma'$ with $m^0 \xrightarrow{\sigma} m$ and $m^0 \xrightarrow{\sigma'} m$, it holds $\psi(\sigma) \Leftrightarrow \psi(\sigma')$. Petri net $N$ is conclusive w.r.t. $\psi$ iff each reachable marking of $N$ is conclusive w.r.t. $\psi$.

Given a conclusive Petri net, there exists a canonical valid abstraction $\text{abs}_N(\psi)$ of $\psi$ w.r.t. $N$. Intuitively, for each marking $m$ we pick an arbitrary run $\sigma$ resulting in $m$. In order to avoid a restriction of this definition to conclusive Petri nets, we build the conjunction of all runs resulting in a given marking.

\textbf{Definition 7 ($N$-abstraction).} We define the $N$-abstraction $\text{abs}_N(\psi)$ of $\psi$ by

$$\text{abs}_N(\psi)(m) := \bigwedge_{\sigma, m^0 \xrightarrow{\sigma} N m} \psi(\sigma).$$

As explained above, the $N$-abstraction of $\psi$ is a valid abstraction, if $N$ is conclusive w.r.t. $\psi$.

\textbf{Lemma 1.} If $N$ is conclusive w.r.t. $\psi$, then $\text{abs}_N(\psi)$ is a valid abstraction of $\psi$ w.r.t. $N$.

\textbf{Proof.} Assume exists $\sigma$ with $m^0 \xrightarrow{\sigma} N m$ and $\psi(\sigma) \neq \text{abs}_N(\psi)(m)$. Consider (1) that $\psi(\sigma)$ holds, and (2) that $\psi(\sigma)$ does not hold.

1. By assumption, $\text{abs}_N(\psi)(m)$ does not hold. Hence, there exists $\sigma'$ with $m^0 \xrightarrow{\sigma'} N m$ such that $\psi(\sigma')$ does not hold. However, this is a direct contradiction to the assumption that $N$ is conclusive w.r.t. $\psi$.

2. By assumption, $\text{abs}_N(\psi)(m)$ holds. Hence, we find that for all $\sigma'$ with $m^0 \xrightarrow{\sigma'} N m$, it holds $\psi(\sigma')$. However, this contradicts the assumption that $\psi(\sigma)$ does not hold.

\hfill $\Box$

We study a certain class of run predicates, namely \textit{safety predicates} [2]. Intuitively, once a run $\sigma$ does not satisfy a safety predicate, it cannot be repaired; that is, $\sigma$ may not be continued by $\sigma'$, such that $\sigma\sigma'$ satisfies $\psi$.

\textbf{Definition 8 (Safety predicate).} Run predicate $\psi$ of $N$ is a safety predicate iff for each $\sigma, \sigma' \in T^*$, it holds: $\neg \psi(\sigma) \Rightarrow \neg \psi(\sigma\sigma').$
Safety predicates are easier to handle during analysis than arbitrary run predicates. For instance, each run $\sigma$ has a minimal prefix w.r.t. a safety predicate $\psi$. Intuitively, the minimal prefix is sufficient to decide if $\sigma$ satisfies $\psi$.

**Definition 9 (Minimal prefix).** Let $N$ be a Petri net, and $\psi$ a run predicate of $N$. Let $\sigma \in T^*$. We define the minimal prefix $\sigma^\psi_{\min}$ of $\sigma$ w.r.t. $\psi$ by

$$
\sigma^\psi_{\min} := \begin{cases} 
\sigma', & \text{if } \sigma' \text{ is the minimal prefix of } \sigma \text{ with } \neg \psi(\sigma') \\
\sigma, & \text{if no such } \sigma' \text{ exists.}
\end{cases}
$$

We define $\psi_{\min}$ by $\psi_{\min}(\sigma) = \psi(\sigma^\psi_{\min})$.

**Lemma 2.** If $\psi$ is a safety predicate, and $\sigma \in T^*$, then it holds $\psi(\sigma) \Leftrightarrow \psi(\sigma^\psi_{\min})$.

**Proof.** We distinguish three cases.

1. Choose $\sigma$, such that $\psi(\sigma)$ does not hold. Then, there exists a minimal prefix $\sigma'$ of $\sigma$, such that $\neg \psi(\sigma')$, and $\sigma' = \sigma^\psi_{\min}$. We find that $\psi(\sigma^\psi_{\min})$ does not hold.
2. Choose $\sigma$, such that $\psi(\sigma)$ holds, and for each prefix $\sigma'$ of $\sigma$, it holds $\psi(\sigma')$.
   Then, $\sigma = \sigma^\psi_{\min}$. We find that $\psi(\sigma^\psi_{\min})$ holds.
3. Choose $\sigma$, such that $\psi(\sigma)$ holds, and there exists a prefix $\sigma'$ of $\sigma$, with $\neg \psi(\sigma')$.
   Because $\sigma'$ is a prefix of $\sigma$, and $\psi$ is a safety predicate, we infer $\neg \psi(\sigma)$. This contradicts the assumption. Hence, no instance of this case exists.

**Corollary 1.** If $\psi$ is a safety predicate, then $\psi \equiv \psi_{\min}$.

### 4 A formal framework for costs

We study costs which are directly induced by the execution path, and abstract from any other incurring costs. As a further restriction, we assume that each action has fixed, known execution costs. Thereby, fixed means that the costs for executing an action do not depend on other occurred actions, or the current state. However, it does not imply that the costs for executing an action are necessarily a single number. They could also be an interval, or an expectation together with a deviation. The set of possible cost values is fixed by means of a cost domain. Formally, a cost domain is simply a partially ordered set. The partial order is useful when studying thresholds. A threshold is a set of cost values, such that for any other cost value it can be determined whether it is above or below the threshold. In order to specify costs, we introduce cost functions. A cost function of an open net $N$ maps a transition of $N$ to a cost value from a cost domain. Formally, we define a cost function as a partial function to later cope with composition: Then, a cost function of $N$ is also a cost function of $N \oplus Q$ for some $Q$. 

Definition 10 (Cost domain, threshold, cost function). A cost domain \([A, \leq]\) consists of a set \(A\) and a partial order \(\leq\) on \(A\). A set \(\triangle \subseteq A\) is a threshold w.r.t. \([A, \leq]\) iff for all \(a \in A\), it holds \(a \leq \triangle\) or \(a \geq \triangle\). Let \([A, \leq]\) be a cost domain, and \(N\) be an open net. A partial function \(f : T_N \rightarrow A\) is a cost function of \(N\) over \([A, \leq]\).

Example 1. \(\mathbb{N}_{0,\infty}\) together with \(\leq\) is a cost domain, \([\mathbb{N}_{0,\infty}, \leq]\). The partial function \(f : T_N \rightarrow \mathbb{N}_{0}\) with domain \(\{B\}\) and \(f(B) = 10\) is a cost function of \(N\) over \([\mathbb{N}_{0,\infty}, \leq]\).

Next, we define costs of a run, and of sets of runs. A cost model specifies aggregation methods for cost values. Thereby, aggregation means the compression of several cost values into one. In our context, we require two types of aggregation: Sequence aggregation and set aggregation. Sequence aggregation yields a single cost value for a sequence of cost values. Set aggregation yields a single cost value for a set of cost values. Intuitively, sequence aggregation will be used to evaluate the costs of a run, whereas set aggregation will yield the costs for a set of runs.

We write down a cost model as an algebraic structure. We reuse the well-studied algebraic structure of a semiring. The carrier set of the semiring corresponds to a cost domain, the operations realize the aggregation. A semiring contains two operations, namely abstract addition and abstract multiplication. To avoid confusion with concrete addition and multiplication operators, we will use the terms set aggregator and sequence aggregator instead.

Definition 11 (Semiring, cost model). A semiring \(M = [A, \bullet, \odot, z, e]\) consists of a carrier set \(A\) with \(z, e \in A\), and two operations \(\bullet\) and \(\odot\), such that

- \(\bullet : A \times A \rightarrow A\) is an associative, commutative binary operation on \(A\) with identity element \(z\),
- \(\odot : A \times A \rightarrow A\) is an associative binary operation on \(A\) with identity element \(e\),
- \(\odot\) distributes over \(\bullet\), and
- \(a \odot z = z \odot a = z\) for all \(a \in A\).

If \(M = [A, \bullet, \odot, z, e]\) is a semiring, and \([A, \leq]\) is a cost domain, then \(M\) is a cost model over \([A, \leq]\). We call \(\bullet\) the set aggregator and \(\odot\) the sequence aggregator of \(M\).

We canonically extend the definition of \(\bullet\) to finite sets: Let \(A = \{a_1, \ldots, a_n\} \subseteq A\), then \(\bullet(A) = a_1 \bullet \ldots \bullet a_n\). If the infinite application of \(\bullet\) is defined, we also define \(\bullet(A)\) for infinite sets \(A \subseteq A\) in the usual way.

Example 2. \(M = [\mathbb{N}_{0,\infty}, \max, +, -\infty, 0]\) is a cost model over \([\mathbb{N}_{0,\infty}, \leq]\).

A cost function and a cost model over the same cost domain are sufficient to determine the costs of a run. We define the costs of a run by first applying \(f\), yielding a sequence of cost values. Second, we apply \(\odot\) to aggregate the sequence into a single cost value. Similarly, we proceed with a set of runs: We aggregate the costs of the single runs with \(\bullet\). We further define the costs of an open net as the costs of its set of terminating runs.
We study their respective costs:
\[ 10 + 0 + 0 = 10. \]

Definition 12 (Costs of runs, sets of runs, and open nets). Let \( f \) be a cost function of open net \( N \) over \([A, \leq]\). Let \( M = [A, \bullet, o, z, e] \) be a cost model over \([A, \leq]\). Let \( \sigma = t_1 \ldots t_n \in T_N^* \). Let \( R \subseteq T_N^* \).

We define the costs of \( \sigma \) w.r.t. \( f \) and \( M \) by \( \langle \sigma \rangle_{f,M} = f_M(t_1) \circ \ldots \circ f_M(t_n) \), where \( f_M : T_N \to A \) with
\[
t(t) \mapsto \begin{cases} f(t), & \text{iff } t \text{ is in the domain of } f \\ e, & \text{otherwise.} \end{cases}
\]

If \( X = \bullet(\{\langle \sigma \rangle_{f,M} | \sigma \in R}\} \) is defined, we define the costs of \( R \) w.r.t. \( f \) and \( M \) by \( \langle R \rangle_{f,M} = X \). If \( Y = \langle \text{TRuns}(N) \rangle_{f,M} \) is defined, we define the costs of \( N \) w.r.t. \( f \) and \( M \) by \( \langle N \rangle_{f,M} = Y \).

Example 3. Consider the composition \( N \oplus Q_1 \) of \( N \) and \( Q_1 \) from Fig. 1. We find that \( \sigma_1 = AFB, \sigma_2 = AGC, \sigma_3 = \sigma_1 D \sigma_2, \) and \( \sigma_4 = \sigma_1 D \sigma_1 \) are runs of \( N \oplus Q_1 \). We study their respective costs: \( \langle \sigma_1 \rangle_{f,M} = 0 + 0 + f(B) = 0 + 0 + 10 = 10 \), \( \langle \sigma_2 \rangle_{f,M} = 0 + 0 + f(B) = 0 + 0 + 0 + 0 = 0 \), \( \langle \sigma_3 \rangle_{f,M} = \langle \sigma_1 \rangle_{f,M} + 0 + \langle \sigma_2 \rangle_{f,M} = 10 + 0 + 10 = 20 \), and \( \langle \sigma_4 \rangle_{f,M} = \langle \sigma_1 \rangle_{f,M} + 0 + \langle \sigma_1 \rangle_{f,M} = 10 + 0 + 10 = 20 \). Consider now the set \( R = \{\sigma_1, \sigma_2, \sigma_3, \sigma_4\} \). We study the costs of \( R \): \( \langle R \rangle_{f,M} = \max(\{\langle \sigma \rangle_{f,M} | \sigma \in R\}) = \max(\{0, 10, 20\}) = 20 \).

A cost specification consists of an open net \( N \), a cost domain, a threshold, a cost function, and a cost model. A partner \( Q \) is cost-bounded, iff the costs for \( N \oplus Q \) are defined. A partner \( Q \) of \( N \) matches the cost specification, iff \( Q \) is a cost-bounded partner, and the costs of \( N \oplus Q \) stay below the threshold.

Definition 13 (Cost specification, matching, cost-bounded). A cost specification \( S = [N, A, \leq, \triangle, f, M] \) consists of an open net \( N \), a cost domain \([A, \leq]\), a threshold \( \triangle \subseteq A \), a cost function \( f \) of \( N \) over \([A, \leq]\), and a cost model \( M \) over \([A, \leq]\).

Let \( \sigma = t_1 \ldots t_n \in T_N^* \). Let \( R \subseteq T_N^* \). Let \( Q \) be a partner of \( N \). Sequence \( \sigma \) matches \( S \) iff \( \langle \sigma \rangle_{f,M} \preceq \triangle \). Set \( R \) is cost-bounded w.r.t. \( S \) if \( \langle R \rangle_{f,M} \) is defined. Set \( R \) matches \( S \) iff \( R \) is cost-bounded w.r.t. \( S \) and \( \langle R \rangle_{f,M} \preceq \triangle \). Partner \( Q \) of \( N \) is cost-bounded w.r.t. \( S \) if \( \langle N \oplus Q \rangle_{f,M} \) is defined. Partner \( Q \) of \( N \) matches \( S \) iff \( Q \) is cost-bounded w.r.t. \( S \) and \( \langle N \oplus Q \rangle_{f,M} \preceq \triangle \). We write \( \text{Partners}_S \) for the set of all partners of \( N \) matching \( S \).

Example 4. Open net \( N \) from Fig. 1, and \([N_{0, \infty}, \leq], \{19\}, f, M \) from the previous examples, form a cost specification \( S_N \). The open nets \( Q_2 \) and \( Q_3 \) from Fig. 1 are cost-bounded w.r.t. \( S_N \). Furthermore, \( Q_2 \) and \( Q_3 \) match \( S_N \), but \( Q_1 \) does not.

Notation 1. Let \( S = [N, A, \leq, \triangle, f, M] \) be a cost specification. Let \( x \in T_N^* \). We sometimes write \( \langle x \rangle_S \) instead of \( \langle x \rangle_{f,M} \).

We introduce two classes of cost specifications: Universal and monotone cost specifications. Intuitively, in a universal cost specification, the costs of each terminating run must stay below the threshold. In a monotone cost specification, a run stays below the threshold if each prefix does.
Definition 14 (Universality, monotony). Let \( S = [N, A, \leq, \triangle, f, M] \) be a cost specification. We call \( S \) universal iff for all \( R \subseteq T_N \), it holds: \( R \) matches \( S \) iff for all \( \sigma \in R \): \( \sigma \) matches \( S \). We call \( S \) monotone iff for all \( \sigma \in T_N \) and \( \ell \in T_N \) it holds: \( \langle \sigma \rangle_S \leq \langle \sigma \ell \rangle_S \).

Lemma 3. Let \( S = [N, A, \leq, \triangle, f, M] \) be a universal, monotone cost specification. Let \( N \) be weakly terminating. Then it holds: \( \text{TRuns}(N) \) matches \( S \) iff for all \( \sigma \in \text{Runs}(N) \): \( \sigma \) matches \( S \).

Proof. “\( \Rightarrow \)”: Assume a run \( \sigma \in \text{Runs}(N) \) not matching \( S \). Then, \( \langle \sigma \rangle_S \not\in \triangle \). Because \( N \) is weakly terminating, \( \sigma \) may be continued to a terminating run \( \sigma' \). Because \( S \) is universal and \( \langle \text{TRuns}(N) \rangle_S \leq \triangle \), it holds \( \langle \sigma' \rangle_S \leq \triangle \). Because \( S \) is monotone and \( \sigma \) is a prefix of \( \sigma' \), it holds \( \langle \sigma \rangle_S \leq \langle \sigma' \rangle_S \), which contradicts the assumption \( \langle \sigma \rangle_S \not\in \triangle \).

“\( \Leftarrow \)”: Assume that \( \text{TRuns}(N) \) does not match \( S \). Because \( S \) is universal, there exists at least one \( \sigma \in \text{TRuns}(N) \), such that \( \sigma \) does not match \( S \), which contradicts the assumption that for all \( \sigma \in \text{Runs}(N) \): \( \sigma \) matches \( S \). \( \square \)

Theorem 1. Let \( S = [N, A, \leq, \triangle, f, M] \) be a universal, monotone cost specification. Let \( Q \) be a partner of \( N \). Then, \( Q \) matches \( S \) iff for all \( \sigma \in \text{Runs}(N \oplus Q) \): \( \sigma \) matches \( S \).

Proof. Because \( Q \) is a partner of \( N \), their composition \( N \oplus Q \) is weakly terminating.

“\( \Rightarrow \)”: If \( Q \) matches \( S \), then \( \text{TRuns}(N \oplus Q) \) matches \( S \). By Lemma 3, for all \( \sigma \in \text{Runs}(N \oplus Q) \): \( \sigma \) matches \( S \).

“\( \Leftarrow \)”: By Lemma 3, \( \text{TRuns}(N \oplus Q) \) matches \( S \). By Def. 13, \( Q \) matches \( S \). \( \square \)

In this paper, we study a specific subclass of universal and monotone cost specifications: The class \( \mathcal{W} \) of worst case total costs. Intuitively, costs are represented as natural numbers. We use addition and maximum as sequence and set aggregator, respectively.

Definition 15 (Worst case total costs). A cost specification \( S = [N, A, \leq, \triangle, f, M] \) is a worst case total costs (WCTC) specification, iff \( A = \mathbb{N}_{0,\infty}, \leq \) is the order on \( \mathbb{N}_{0,\infty}, M = [\mathbb{N}_{0,\infty}, \max, +, -\infty, 0] \). We denote the set of all WCTC specifications by \( \mathcal{W} \).

Example 5. Example cost specification \( S_N \) is an element of \( \mathcal{W} \).

WCTC specifications are monotone and universal: The addition on natural numbers is monotone. Hence, adding a transition never decreases the costs. The supremum of a set of natural numbers is always greater than or equal to the elements in the set. Hence, the costs of a run never rise above the supremum.

Lemma 4 (Monotony and universality). If \( S \in \mathcal{W} \), then \( S \) is monotone and universal.

Proof. \( + \) is monotone, hence, \( S \) is monotone. \( \max \) picks the greatest element from a set, and \( \leq \) is a total order. \( \square \)
5 Computing representatives for worst case total costs

In order to integrate costs into existing analysis techniques for stateful services, we advise to substitute the subject open net \( N \) by a representative \( R \) w.r.t. a cost specification \( S \). A representative w.r.t. a given cost specification \( S \) is an open net satisfying two properties: First, \( N \) and \( R \) have the same set of \( S \)-partners. Second, each partner of \( R \) is a \( S \)-partner of \( R \). Therefore, \( R \) is more restrictive than \( N \), and the sets \( \text{Partners}_S(N) \) and \( \text{Partners}(R) \) are equal.

**Definition 16 (Representative).** Open net \( R \) is a \( S \)-representative of \( N \), iff

1. \( \text{Partners}_S(N) = \text{Partners}_S(R) \), and
2. \( \text{Partners}_S(R) = \text{Partners}(R) \).

We denote the set of all representatives of \( N \) by \( \text{Rep}_S(N) \).

As a result, we may analyze \( R \) instead of \( N \), yielding the same results.

**Corollary 2.** Let \( N \) be an open net and \( R \in \text{Rep}_S(N) \). Then, \( \text{Partners}_S(N) = \text{Partners}(R) \).

In Sect. 5, we tackle the problem to compute a canonical \( S \)-representative. Thereby, we restrict ourselves to a class of cost specifications, which is monotone and universal.

We devote the remainder of this section to the computation of a \( S \)-representative of an open net \( N \) for some given \( S \in \mathcal{W} \). First, we solve the problem for a \( S \)-conclusive open net. Second, we explain how an arbitrary open net \( N \) may be transformed into a \( S \)-conclusive open net.

5.1 Representatives for conclusive open nets

Consider a \( S \)-conclusive open net \( N \). Then, we may define a canonical representative \( N_S \) based on the \( N \)-abstraction \( \text{abs}_N(S) \) of \( S \). Intuitively, we modify the final markings by building the intersection with \( \text{abs}_N(S) \). The intersection is well defined because \( \text{abs}_N(S) \) is a state predicate of \( N \), and thus a set of markings.

**Definition 17 (Canonical representative).** The canonical \( S \)-representative \( N_S \) of a conclusive open net \( N \) is the open net \( N_S := [P, T, V, m^0, M, I, O, \lambda] \) where \( M = M^f \cap \text{abs}_N(S) \).

**Example 6.** Open net \( N_1 \) from Fig. 2 is a \( S_N \)-conclusive variant of open net \( N \): Place \( s \) may be used to decide whether the costs are above or below the threshold \( \Delta \). Namely, more than one token proves that the costs rose above the threshold. Adjusting the final markings accordingly yields \( N_1 \).

**Lemma 5.** \( \text{Partners}_S(N) = \text{Partners}_S(N_S) \).

**Proof.** We show both directions separately. In both cases, we choose a partner \( Q \). Then, each run \( \sigma \in N \otimes Q \) is a run of \( N_S \otimes Q \), and vice versa.
Example 7. Example services \(Q_2\) and \(Q_3\) (Fig. 1) are partners of \(N\) (Fig. 1), matching \(S_N\). \(Q_2\) never sends \(\text{NOW}\), and \(Q_3\) (Fig. 1) sends it once. Hence, in any reachable marking \(m\) with either partner, \(m(p_{S_2}) < 2\). Therefore, we retain the complete behavior. Hence, \(Q_2\), and \(Q_3\) match \(S_N\).

**Lemma 6.** \(\text{Partners}_S(N_S) = \text{Partners}(N_S)\).

**Proof.** The direction \(\text{Partners}_S(N_S) \subseteq \text{Partners}(N_S)\) trivially holds. We show \(\text{Partners}_S(N_S) \supseteq \text{Partners}(N_S)\). Choose some partner \(Q\) of \(N_S\). Then, for each run \(\sigma\) of \(N_S\), there exists \(\sigma'\), such that \(\sigma\sigma'\) is a terminating run. Let \(\sigma\sigma'\) result in \(m\). Because \(S\) is a safety predicate, \(\text{abs}_N(S)\) is a valid abstraction of \(S\) w.r.t. \(N_S\). Hence, \(\sigma\sigma'\) matches \(S\), because \(\text{abs}_N(S)(m)\).

**Example 8.** Assume an arbitrary partner \(Q\) of \(N_1\) from Fig. 2. It is obvious that any final marking of \(N_1\) is a marking where the costs are below the threshold.
Corollary 3. \( N_S \in \text{Rep}(N) \).

Computing \( N_S \) therefore boils down to computing \( \text{abs}_N(S) \). In the next section, we solve this problem by providing a transformation procedure for arbitrary open nets with the following result: First, the result is a conclusive open net \( \text{mod}_S(N) \). As a second result, the procedure yields \( \text{abs}_{\text{mod}_S(N)}(S) \).

5.2 Making open nets conclusive

Next, we study arbitrary open nets. Thereby, we exploit the fact that WCTC-specifications correspond to safety predicates: Once the costs of a run rise above the threshold, they stay above the threshold forever. The idea is to construct a new net \( \text{mod}_S(N) \), such that \( \text{mod}_S(N) \) is conclusive, and \( \text{abs}_{\text{mod}_S(N)}(S) \) is obvious. One solution is to add a place \( p_S \) to \( N \), logging the costs. Here, logging means that the number of tokens on \( p_S \) equals the costs of the current run. The resulting net is conclusive: Consider a run \( \mathcal{\Pi} \) resulting in marking \( m \). Then, \( m(p_S) \leq \Delta \) indicates that the costs of \( \mathcal{\Pi} \) are acceptable. If in contrast \( m(p_S) > \Delta \), then the costs of \( \mathcal{\Pi} \) are above the threshold. This observation also yields \( \text{abs}_{\text{mod}_S(N)}(S) \).

Definition 18. The \( S \)-modification \( \text{mod}_S(N) \) of \( N \) is the open net \( \text{mod}_S(N) := [P \cup \{p_S\}, T, V \cup V', m^0, M^f + \text{Bags}(\{p_S\}), I, O, \lambda] \) where for all \( t \in T \): \( V'(t, p_S) = f(t) \), and \( V'(p_S, t) = 0 \). We define the canonical state predicate \( \text{pred}(S) \) by \( \text{pred}(S)(m) \rightarrow m(p_S) \leq \Delta \).

Example 9. The \( S_N \)-modification \( \text{mod}_S(N) \) is the open net \( N_2 \) in Fig. 2: We added a place \( s \), and transition \( B \) produces \( f(B) = 10 \) tokens on \( s \). Place \( s \) does not constrain any transition of \( N_2 \). Hence, adding \( s \) does not change the set of runs. Because the final markings of \( N_2 \) are indifferent to the number of tokens on \( s \), the set of terminating runs does not differ either.

Lemma 7. \( \text{pred}(S) \equiv \text{abs}_{\text{mod}_S(N)}(S) \).

Proof. Assume two runs \( \sigma, \sigma' \) of \( \text{inner}(\text{mod}_S(N)) \) resulting in marking \( m \). As a matter of fact, \( \langle \sigma \rangle_S = \langle \sigma' \rangle_S = m(p_S) \). Hence, \( \text{pred}(S)(\sigma) \Leftrightarrow \text{pred}(S)(\sigma') \). Hence, we find:
\[
\bigwedge_{\sigma : m^0 \xrightarrow{\text{inner}(N)} m} \sigma \Leftrightarrow m(p_S) \leq \Delta.
\]
\( \square \)

Lemma 8. \( \text{mod}_S(N) \) is conclusive.

Proof. It is sufficient to know the costs of the current run to decide matching of the run. This cost value is immediately represented by the number of tokens on \( p_S \).
\( \square \)

However, this solution has a disadvantage. In general, \( \text{inner}(\text{mod}_S(N)) \) is unbounded, that is, has infinitely many reachable markings. This even occurs when \( N \) is bounded, that is, finite state.
Example 10. The inner of $N_2$ (Fig. 2) is unbounded, because $K$ may fire arbitrarily often.

We overcome this problem by application of a standard Petri net technique. We introduce a complementary place $p_S$ for $p_S$. Intuitively, a transition consuming $n$ tokens from $p_S$ simultaneously produces $n$ tokens on $p_S$. Likewise, a transition producing $n$ tokens on $p_S$ simultaneously consumes $n$ tokens from $p_S$. The initial marking of $p_S$ relies on the threshold $\Delta$, and the costs $f_{\text{max}}$ of the most expensive transition. Initially, there are $\Delta + f_{\text{max}}$ tokens on $p_S$. This procedure cuts away behavior: Once the threshold has been breached, no more transitions with costs higher than zero may occur. However, this is acceptable, because $S$ is a safety predicate: It is sufficient to keep the minimal prefix of each run.

Example 11. Figure 2 shows $N_3$, a variant of $N_2$ and $N_1$ (Fig. 2). The idea is that place $s$ in $N_3$ is bounded by introducing an additional place $u$ with 20 initial tokens. We find that $B$ may only occur twice in inner($N_3$) instead of arbitrarily often, as in $N$ or $N_2$. However, after the second execution of $B$, the costs have been risen above the threshold. Hence, the minimal prefixes are kept, and no valuable information is lost.

6 Related work

In [1], the authors extend discrete-time Petri nets with a cost model, studying the issue of minimal cost reachability and coverability. We do not study the minimal costs for reaching a certain state, but the maximal costs to terminate. Additionally, we study models of services in contrast to closed systems. Annotating Petri nets with costs is similar to using Weighted finite automata (WFA) (see e.g. [4]). We could also approach the topic with a special class of WFA, yielding similar results. Zeng et al. [12] use integer programming to find an optimal composition of atomic tasks each implemented by a web service. The services do not communicate based on its state and therefore do not influence each others costs. De Paoli et al. [6] propose a similar approach for WS-BPEL [3] processes. Both approaches work on well-structured services, whereas we support arbitrary services.

7 Conclusion and future work

In this paper, we introduced a framework to enhance purely behavioral models of services with cost specifications. Thereby, we separate the actual annotation with costs from the aggregation of costs along runs and sets of runs. For a class of cost specifications, we explain a transformation procedure, yielding an equivalent, purely behavioral model. For future work, we plan to investigate additional classes of cost specifications. We plan to generalize our result for the class of all universal, monotone cost specifications. Additionally, we believe that other classes are of interest, for example average costs. Our results to find a minimal cost threshold for a given service, yielding cost minimal partners, are preliminary [10]. We
implemented the transformation procedure in a prototype called Tara\(^1\). However, a comprehensive case study for this approach is still missing.

References


---

\(^{1}\) Tara is available at http://service-technology.org/tara
Abstract. Conformance checking techniques can be used to diagnose differences between observed behavior and modeled behavior. Although these techniques can be used to measure the degree of conformance of a running service based on recorded event data (e.g., messages or transaction logs) and its specification, their application may produce “false negatives” because a private view (i.e., an implementation) that accords with its specification may deviate significantly. The implementation may reorder some activities without introducing any problems, yet traditional conformance checking would penalize such changes unjustifiably. To overcome this problem, we present a novel approach that determines a best matching private view. We show that among the infinitely many accordant private views, there is a canonical best matching private view. Although the current implementation and experiments are limited to acyclic service models, the approach can also be applied to cyclic service models.

1 Introduction

Service-oriented computing (SOC) [17] aims at building complex systems by aggregating less complex, independently-developed building blocks called services. A service encapsulates a business functionality and has an interface to interact with its environment—that is, other services—via asynchronous message passing. The service-oriented paradigm enables enterprises to publish their services via the Internet. These services can then be automatically found and used by other enterprises. However, in practice, enterprises usually cooperate only with enterprises they already know. Therefore, a more pragmatic approach is often used instead: The involved parties specify a public version of the overall service, which serves as a contract [1,5]. Later on, each party implements (refines) its share of the contract. A party’s share of the contract—that is, the public view—and the implementation thereof—that is, the private view—may differ significantly but the overall implementation has to conform to the contract. Correctness of a contract (i.e., the possibility to always terminate) has been formalized by the accordance relation [19] between a public view and a private view: if every private view accords with its public view, then the correctness of the contract is preserved and the overall implementation conforms to the contract. Accordance thereby guarantees that any environment that cooperates with the public view can cooperate with the private view.
As an example, consider the public view in Fig. 1a, which is modeled as an open net—that is, a Petri net extended with interface places\(^1\). The open net either sends message \(b\) and then receives \(d\) or sends message \(a\) and then receives \(c\) or \(d\). A possible private view is shown in Fig. 1b. It is derived from the public view by parallelizing the sending and receiving of messages. In contrast to the public view, the private view can, therefore, receive \(c\) after having sent \(b\). Open net Private accords with open net Public. Intuitively, every cooperating environment of Public knows by receiving either \(a\) or \(b\) whether Public is in the left or the right branch. Therefore, no cooperating environment of Public will send \(c\) after having received \(b\), as otherwise the cooperation may get stuck. Public may operate in such an environment. In fact, it even allows for environments that send \(c\) after having received \(b\).

Although accordance can be checked efficiently, the approach can hardly be used in practice, because the accordance check assumes that the public and the private view of

\(^1\) Throughout this paper, we assume public views to be given as open nets.
a party are given as formal models that do not change over time. However, it is often not realistic to assume that all parties will indeed have an up-to-date formal model of their private view. Even if they have a formal model of the private view, it can differ significantly from the actual implementation: services may have been implemented incorrectly or change over time. Nevertheless, most implementations provide some kind of recorded behavior (also referred to as event log, transaction log or audit trail) \[3\]. Therefore, instead of checking accordance of the public and private view, we check whether event logs of the private view conform to the public view. Figure 2 illustrates our approach for conformance checking in the contract setting.

We illustrate the idea using open net Public and the event log \(L\) in Fig. 3c, which represents the recorded behavior of the implementation of Public. \(L\) contains information of 120 traces, partitioned into three cases. A trace is a sequence of messages sent or received by the implementation. We assume that each event \(x\) in a trace of a log corresponds to the sending or receiving of \(x\) of the environment. We can model this environment of an open net by adding to each \(x\)-labeled input place an \(x\)-labeled transition that produces tokens on this place and for each \(x\)-labeled output place an \(x\)-labeled transition that consumes tokens from this place. All other transitions of this environment are internal and, therefore, labeled by \(\tau\). Figure 3a illustrates this construction for the public view Public; for convenience, we omit all \(\tau\) labels of transitions.

To check whether \(L\) conforms to the (environment of the) public view Public, we need to replay the traces of \(L\) on the model in Fig. 3a. More precisely, we align each trace in \(L\) to a trace (i.e., a firing sequence) of the model in Fig. 3a. Some example alignments for \(L\) and the environment of Public are:

\[
\gamma_1 = d, a, d, a, d, \tau, \tau \\
\gamma_2 = d, a, d, a, d, a, d, \tau, \tau \\
\gamma_3 = c, d, a, d, a, \tau, \tau, \tau, \tau, \tau
\]
The top row of each alignment corresponds to “moves in the log” and the bottom two rows correspond to “moves in the model”. There are two bottom rows because multiple transitions may have the same label; the upper bottom row consists of transition labels, and the lower bottom row consists of transitions. If a move in the log cannot be mimicked by a move in the model, then a “\(\triangleright\) (no move)” appears in the upper bottom row. For example, in \(\gamma_2\) the model in Fig. 3a cannot do the last c-move, because c is not connected to the locally enabled transition \(t_2\). If a move in the model cannot be mimicked by a move in the log, then a “\(\triangleright\) (no move)” appears in the top row. For example, all “silent moves” (occurrences of \(\tau\)-labeled transitions) in the model in Fig. 3a cannot be mimicked by L. Moreover, L did not do a d-move in \(\gamma_2\) whereas the model in Fig. 3a has to make this move to reach the end.

Informally, conformance checking of an event log \(L\) and a public view \(N\) relies on “how good” each case in \(L\) can be replayed in the environment of \(N\). Therefore, the smaller the number of mismatches in an alignment of a case is, the better this case can be replayed. A mismatch is a move in the log which cannot be mimicked by the model, or a non-silent move in the model which cannot be mimicked by the log. Clearly, the more traces we can replay on the model the better the implementation conforms to the public view. However, even an implementation that accords with its public view may allow for traces that cannot be replayed on the public view, because the accordance relation allows parties to reorder activities of their share, for instance. As an illustration, consider again the private view in Fig. 1b. We can replay the event log \(L\) on the model of the environment of this open net, which is depicted in Fig. 3b. Some resulting alignments are:

\[
\gamma_4 = \begin{array}{c|c|c|c|c|c|c|c}
\triangleright\triangleright & a & \triangleright & d & \triangleright & \triangleright & \\
\tau & \tau & \tau & \tau & \tau & \tau & \\
t_0 & t_1 & t_2 & t_3 & t_4 & t_5 & \\
\end{array}
\quad \gamma_5 = \begin{array}{c|c|c|c|c|c|c|c}
\triangleright\triangleright & b & \triangleright & c & \triangleright & \\
\tau & \tau & \tau & \tau & \tau & \\
t_0 & t_1 & t_2 & t_3 & t_4 & t_5 & \\
\end{array}
\quad \gamma_6 = \begin{array}{c|c|c|c|c|c|c|c}
\triangleright & d & \triangleright & \triangleright & a & \triangleright & \triangleright & \\
\tau & \tau & \tau & \tau & \tau & \tau & \\
t_0 & t_1 & t_2 & t_3 & t_4 & t_5 & \\
\end{array}
\]

The example clearly shows that, in general, it is not sufficient to check conformance of an event log and the model of the public view. Checking conformance on the public view may generate “false negatives”, i.e., acceptable behavior may be diagnosed as non-conforming. As there may exist a private view that accords with the public view such that the conformance check with that model gives a better result, we need to check conformance of a log with all private views that accord with the public view. The challenge thereby is that there exist infinitely many such private views. In this paper, we investigate this challenge and present an approach to determine a best matching private view for a given event log and a public view.

The remainder is organized as follows. More background information is provided in Sect. 2. Section 3 shows the existence of a canonical best matching private view. Experimental results in Sect. 4 validate our approach. In Sect. 5, we review related work, and Sect. 6 concludes the paper.

2 Background

In this section, we provide the basic notions of Petri nets and open nets for modeling services and formalize private view conformance. Suitability of open nets as service
We write transition labels beside as a basic model, we use place/transition Petri nets extended with a set of final markings and transition labels.

**Definition 1 (Net).** A net \( N = (P, T, F, m_N, \Omega) \) consists of a finite set \( P \) of places, a finite set \( T \) of transitions such that \( P \) and \( T \) are disjoint, a flow relation \( F \subseteq (P \times T) \cup (T \times P) \), an initial marking \( m_N \), where a marking \( m \in B(P) \) is a multiset over \( P \), and a set \( \Omega \) of final markings.

A labeled net is a net \( N \) together with an alphabet \( \mathcal{A} \) of actions and a labeling function \( l \in T \to \mathcal{A} \cup \{\tau\} \), where \( \tau \notin \mathcal{A} \) represents an invisible, internal action.

Graphically, a circle represents a place, a box represents a transition, and the directed arcs between places and transitions represent the flow relation. A marking is a distribution of tokens over the places. Graphically, a black dot represents a token. We write transition labels beside \( \tau \) into the respective boxes.

Let \( x \in P \cup T \) be a node of a net \( N \). As usual, \( x^\bullet = \{y | (y, x) \in F\} \) denotes the preset of \( x \) and \( x^\bullet = \{y | (x, y) \in F\} \) the postset of \( x \). We interpret presets and postsets as multisets when used in operations also involving multisets. For markings, we define + and - for the sum and the difference of two markings in the standard way.

The behavior of a net \( N \) relies on changing the markings of \( N \) by firing transitions of \( N \). A transition \( t \in T \) is enabled at a marking \( m \), denoted by \( m \xrightarrow{t} \), if for all \( p \in \cdot \bullet t \), \( m(p) > 0 \). If \( t \) is enabled at \( m \), it can fire, thereby changing the marking \( m \) to a marking \( m' = m - \bullet t + \bullet \). The firing of \( t \) is denoted by \( m \xrightarrow{t} m' \); that is, \( t \) is enabled at \( m \) and firing it results in \( m' \).

The behavior of \( N \) can be extended to sequences: \( m_1 \xrightarrow{t_1} \ldots \xrightarrow{t_{k-1}} m_k \) is a run of \( N \) if for all \( 0 < i < k \), \( m_i \xrightarrow{t_i} m_{i+1} \). A marking \( m' \) is reachable from a marking \( m \) if there exists a (possibly empty) run \( m_1 \xrightarrow{t_1} \ldots \xrightarrow{t_{k-1}} m_k \) with \( m = m_1 \) and \( m' = m_k \); for \( w = (t_1 \ldots t_{k-1}) \), we also write \( m \xrightarrow{w} m' \). Marking \( m' \) is reachable if it is reachable from \( m_N \). The set \( M_N = \{m' | \exists w : m_N \xrightarrow{w} m'\} \) represents all reachable markings of \( N \).

In the case of labeled nets, we lift runs to traces: If \( m \xrightarrow{w} m' \) and \( v \) is obtained from \( w \) by replacing each transition by its label and removing all \( \tau \)-labels, we write \( m \xrightarrow{v} m' \). For example, if \( w = (t_1 t_2 t_3) \), \( l(t_1) = a \), \( l(t_2) = \tau \), and \( l(t_3) = b \), and \( m \xrightarrow{a b} m' \), then \( m \xrightarrow{a \tau b} m' \) with \( v = (a a b) \).

A net \( N \) is bounded if there exists a bound \( b \in \mathbb{N} \) such that for all reachable markings \( m \in M_N \) and all places \( p \in P \), \( m(p) \leq b \). A reachable marking \( m \notin \Omega \) of \( N \) is a deadlock if no transition \( t \in T \) of \( N \) is enabled at \( m \). If \( N \) has no deadlock, then it is deadlock free. A net is weakly terminating if from every reachable marking it is always possible to reach a final marking.
2.2 Open Nets

We model services as open nets [21,9], thereby restricting ourselves to the communication protocol of a service. In the model, we abstract from data and identify each message by the label of its message channel. An open net extends a net by an interface. An interface consists of two disjoint sets of input and output places corresponding to asynchronous input and output channels. In the initial marking and the final markings, interface places are not marked. An input place has an empty preset, and an output place has an empty postset.

Definition 2 (Open net). An open net \( N \) is a tuple \( (P, T, F, m_N, I, O, \Omega) \) with

- \( (P \cup I \cup O, T, F, m_N, \Omega) \) is a net such that \( P, I, O \) are pairwise disjoint;
- for all \( p \in I \cup O, m_N(p) = 0 \), and for all \( m \in \Omega \) and \( p \in I \cup O, m(p) = 0 \);
- the set \( I \) of input places satisfies for all \( p \in I, \bullet p = \emptyset \) and
- the set \( O \) of output places satisfies for all \( p \in O, p^* = \emptyset \).

Open net \( N \) is sequentially communicating if each transition is connected to at most one interface place. If \( I = O = \emptyset \), then \( N \) is a closed net. Two open nets are interface-equivalent if they have the same sets of input and output places.

Graphically, we represent an open net like a net with a dashed frame around it. The interface places are positioned on the frame.

For the composition of open nets, we assume that the sets of transitions are pairwise disjoint and that no internal place of an open net is a place of any other open net. In contrast, the interfaces overlap intentionally. We require that all communication is bilateral and directed; that is, every shared place \( p \) has only one open net that sends into \( p \) and one open net that receives from \( p \). We refer to open nets that fulfill these properties as composable. We compose two composable open nets \( N_1 \) and \( N_2 \) by merging shared interface places and turn these places into internal places. The definition of composable thereby guarantees that an open net composition is again an open net (possibly a closed net).

Definition 3 (Open net composition). Open nets \( N_1 \) and \( N_2 \) are composable if \( (P_1 \cup T_1 \cup I_1 \cup O_1) \cap (P_2 \cup T_2 \cup I_2 \cup O_2) = (I_1 \cap O_2) \cup (I_2 \cap O_1) \). The composition of two composable open nets \( N_1 \) and \( N_2 \) is the open net \( N_1 \oplus N_2 = (P, T, F, m_N, \Omega, I, O) \) where

- \( P = P_1 \cup P_2 \cup (I_1 \cap O_2) \cup (I_2 \cap O_1) \), \( T = T_1 \cup T_2 \), \( F = F_1 \cup F_2 \);
- \( m_N = m_{N_1} + m_{N_2} \), \( I = (I_1 \cup I_2) \setminus (O_1 \cup O_2) \), \( O = (O_1 \cup O_2) \setminus (I_1 \cup I_2) \);
- \( \Omega = \{m_1 + m_2 \mid m_1 \in \Omega_1, m_2 \in \Omega_2\} \).

We want the composition of a set of services to be correct. Correctness refers to boundedness and weak termination. A user that communicates with a service such that the composition is correct can be seen as a controller of this service.

Definition 4 (Controller). Let \( b \in \mathbb{N} \). An open net \( C \) is a \( b \)-controller of an open net \( N \) if the composition \( N \oplus C \) is a closed net, \( b \)-bounded, and weakly terminating.

In the remainder of the paper, we abstract from the actual bound chosen and, therefore, use the term controller rather than \( b \)-controller for convenience.
2.3 Private View Conformance

We see a contract as a closed net $N$, where every transition is assigned to one of the involved parties $X_1, \ldots, X_k$. We impose only one restriction: if a place is accessed by more than one party, it should act as a directed bilateral communication place. This restriction reflects the fact that a party’s public view of the contract is a service again. A contract $N$ can be cut into parts $N_1, \ldots, N_k$, each representing the agreed public view of a single party $X_i$ ($1 \leq i \leq k$). Hence, we define a contract as the composition of the open nets $N_1, \ldots, N_k$. For instance, Fig. 2 illustrates a contract involving four parties.

**Definition 5 (Contract).** Let $\mathcal{X} = \{X_1, \ldots, X_k\}$ be the set of parties and let $\{N_1, \ldots, N_k\}$ be a set of pairwise interface-compatible open nets such that $N = N_1 \oplus \cdots \oplus N_k$ is a closed net. Then, $N$ is a contract for $\mathcal{X}$. For $i = 1, \ldots, k$, open net $N_i$ is the public view of $X_i$ in $N$ and open net $N_i^{-1} = \bigoplus_{j \neq i} N_j$ is the environment of $X_i$ in $N$.

Each Party $X_i$ can independently substitute its public view $N_i$ by a private view $N_i'$ if the environment of $X_i$ cannot distinguish between $N_i$ and $N_i'$ [13], which is formalized by the accordance relation [19].

**Definition 6 (Accordance).** Let $N_i$ and $N_i'$ be interface-equivalent open nets. Open net $N_i'$ accords with open net $N_i$, denoted by $N_i' \sqsubseteq_{\text{acc}} N_i$, if every controller of $N_i$ is also a controller of $N_i'$.

Sending or receiving a message is an activity. Let $\mathcal{A}$ denote the set of all activities. We define an event log as a multiset of traces over $\mathcal{A}$. Each trace describes the life-cycle of a particular case in terms of the activities executed.

**Definition 7 (Event log).** An event log $L_i$ of the observed behavior of party $X_i$ in contract $N$ is a multiset of traces over $\mathcal{A}$, i.e., $L_i \in \mathcal{B}(\mathcal{A}^*)$.

We use labeled nets to relate event logs to process models. The behavior of a labeled net $N$ is described by the runs of $N$ leading from the initial marking to a final marking.

**Definition 8 (Traces of a labeled net).** Let $N = (P, T, F, m_N, \Omega, l)$ be a labeled net. The set of final runs of $N$ is $R(N) = \{\sigma \in T^* \mid \exists m_f \in \Omega : m_N \xrightarrow{\sigma} m_f\}$, and $Tr(N) = \{\sigma \in \mathcal{A}^* \mid \exists m_f \in \Omega : m_N \xrightarrow{\sigma} m_f\}$ is the set of final traces.

For conformance checking of party $X_i$, we compare the observed behavior (event log $L_i$) with the modeled behavior ($N_i$ or $N_i'$). We can take two viewpoints depending on what/when events are recorded in $L_i$. If events are recorded when party $X_i$ consumes a message from $N_i^{-1}$ or produces a message for $N_i^{-1}$, then we can use the synchronous environment $\text{env}^*(N_i)$ for conformance checking. Here, we label each transition with the adjacent interface places—if possible—and remove the interface places. To simplify the labeling of transitions connected to interface places, we only consider sequentially communicating nets. That way, each transition is labeled by a single label rather than by a set of labels. This restriction is not significant, as every open net can be transformed into an equivalent sequentially communicating open net [9].
Definition 9 (Synchronous environment). The synchronous environment of a sequentially communicating open net \( N = (P, T, F, m_N, \Omega, I, O) \) is the labeled net \( \text{env}^s(N) = (P, T, F \cap ((P \times T) \cup (T \times P)), m_N, \Omega, I, O) \) with \( l(t) = p \) where \( p \) is the unique interface place \( p \in I \cup O \) adjacent to \( t \in T \), or \( l(t) = \tau \) if no such adjacent interface place exists.

If events are recorded when the environment \( N_i^{-1} \) of party \( X_i \) consumes a message from party \( X_j \) or produces a message for party \( X_i \), then we can use the asynchronous environment \( \text{env}^a(N_i) \) for conformance checking. The net \( \text{env}^a(N) \) is a net that can be constructed from \( N \) by adding to each interface place \( p \in I \cup O \) a \( p \)-labeled transition \( t^p \) in \( \text{env}^a(N) \). Intuitively, the construction translates the asynchronous interface of \( N \) into a synchronous interface with unbounded buffers described by the transition labels of \( \text{env}^a(N) \).

Definition 10 (Asynchronous environment). The asynchronous environment of an open net \( N = (P, T, F, m_N, I, O, \Omega) \) is the labeled net \( \text{env}^a(N) = (P \cup I \cup O, T \cup T', F \cup F', m_N, \Omega, I, O) \) where \( T' = \{ t^x \mid x \in I \cup O \} \), \( F' = \{ (t^x, x) \mid x \in I \} \cup \{(x, t^x) \mid x \in O \} \), \( l(t) = x \) for \( t^x \in T' \), and \( l(t) = \tau \) for \( t \in T \).

Figures 3a and 3b show the asynchronous environments of the open nets Private and Public from Figs. 1a and 1b. A transition label is depicted inside a transition with bold font to distinguish it from the transition’s identity.

Thus, the choice of environment depends on what is actually logged. In the remainder, we will abstract from these subtle differences and simply write \( \text{env}^a(N) \).

To check conformance, we need to align traces in the event log to traces of the service (environment): that is, we need to relate “moves” in the log to “moves” in the model. However, there may be some moves in the log that cannot be mimicked by the model, and vice versa. For convenience, we introduce the set \( A_L = A \cup \{ \Rightarrow \} \) where \( x \in A_L \setminus \{ \Rightarrow \} \) refers to “move \( x \) in the log” and \( \Rightarrow \in A_L \) refers to “no move in the log”. Similarly, for a labeled net \( N \), we introduce the set \( A_N = \{ (a, t) \in (A \cup \{ \tau \}) \times T \mid l(t) = a \} \cup \{ \Rightarrow \} \) where \( (a, t) \in A_N \) refers to “move \( a \) in the model” and \( \Rightarrow \in A_N \) refers to “no move in the model”.

A “move \( \tau \) in the model” \( (\tau, t) \) is a silent move, as it is only observable by party \( X_i \).

Definition 11 (Alignment). For an event log \( L \) and a labeled net \( N \), one move in an alignment is represented by a pair \( (x, y) \in A_L \times A_N \) such that

- \( (x, y) \) is a move in the log if \( x \in A \) and \( y = \Rightarrow \),
- \( (x, y) \) is a move in the model if \( x = \Rightarrow \) and \( y \in A_N \setminus \{ \Rightarrow \} \),
- \( (x, y) \) is a move in both if \( x \in A \) and \( y \in A_N \setminus \{ \Rightarrow \} \),
- \( (x, y) \) is an illegal move \( x = \Rightarrow \) and \( y = \Rightarrow \).

We refer to a move in the model \( (x, (a, t)) \) with \( a = \tau \) as a silent move. \( A_{LN} = \{ (x, y) \in A_L \times A_N \mid x \neq \Rightarrow \lor y \neq \Rightarrow \} \) is the set of all legal moves.

An alignment of \( \sigma \in L \) and \( w \in R(N) \) is a sequence \( \gamma \in A_{LN}^* \) such that the projection on the first element (ignoring \( \Rightarrow \)) yields \( \sigma \) and the projection on the second element (ignoring \( \Rightarrow \)) yields \( w \). The set of alignments for \( \sigma \) in \( N \) is \( T_{\sigma, N} = \{ \gamma \in A_{LN}^* \mid \exists w \in R(N) : \gamma \) is an alignment of \( \sigma \) and \( w \}. \)
Given a log trace, there may be many possible alignments. To measure the quality of an alignment, we define a *distance function* on legal moves.

**Definition 12 (Distance function).** A distance function \( \delta : A_{L,N} \rightarrow \mathbb{N} \) associates costs to legal moves in an alignment. We define a standard distance function \( \delta_S \) as:

\[
\delta_S(a, \triangleright\triangleright) = 1; \quad \delta_S(\triangleright\triangleright, (b, t)) = 1, \text{ for } b \neq \tau; \quad \delta_S(\triangleright\triangleright, (\tau, t)) = 0; \quad \delta_S(a, (b, t)) = 0, \text{ for } a \neq \triangleright\triangleright \text{ and } a = b; \quad \delta_S(a, (b, t)) = \infty, \text{ for } a \neq \triangleright\triangleright \text{ and } a \neq b.
\]

We generalize a distance function \( \delta \) to alignments by taking the sum of the costs of all individual moves:

\[
\delta(\gamma) = \sum_{(x,y) \in \gamma} \delta(x,y).
\]

In \( \delta_S \), only moves where log and model agree on the activity, and silent moves of the model have no associated costs. Moves in only the log or model have cost 1, moves where both log and model make a move but disagree on the activity have high costs; thereby, \( \infty \) should be read as a number large enough to discard the alignment. Note that \( \delta_S \) is just an example cost function; various cost functions can be defined.

Thus far, we considered a *specific* trace of the model. However, our goal is to identify for each log trace the *best matching* trace of the model. Therefore, we define the notion of an *optimal alignment*.

**Definition 13 (Optimal alignment).** An alignment \( \gamma \in \Gamma_{\sigma,N} \) is optimal for a log trace \( \sigma \in L \) and a labeled net \( N \) if for any \( \gamma' \in \Gamma_{\sigma,N} : \delta(\gamma') \geq \delta(\gamma) \).

If \( R(N) \) is not empty, there is at least one (optimal) alignment for any given log trace \( \sigma \). However, there may be multiple optimal alignments for \( \sigma \). Since our goal is to align traces in the event log to traces of the model, we nondeterministically select an arbitrary optimal alignment. Therefore, we can construct a function \( \lambda_N \) that provides an “oracle”.

**Definition 14 (Oracle).** Given a log trace \( \sigma \) and a labeled net \( N \), the oracle \( \lambda_N \) produces one optimal alignment \( \lambda_N(\sigma) \in \Gamma_{\sigma,N} \).

The alignments produced by the “oracle” \( \lambda_N \) can be used to quantify conformance of a log \( L \) and a model \( N \). Conformance checking involves the interplay of four orthogonal dimensions: fitness, precision, generalization, and simplicity [2]. Fitness indicates how much of the behavior in the event log is captured by the model. Precision indicates whether the model is not too general. To avoid “underfitting” we prefer models with minimal behavior to represent as closely as possible the behavior seen in the event log. Generalization penalizes overly precise models which “overfit” the given log, and simplicity refers to models minimal in structure, which clearly reflect the log’s behavior.

In the remainder, we abstract from the dimensions involved in conformance checking: we assume a function \( conf \) that computes the conformance of an event log \( L \) and a labeled net \( N \) based on the alignments produced by the oracle \( \lambda_N \); that is, \( conf(L, N) \) yields a number between 0 (poor conformance) and 1 (perfect conformance) [2]. We define private view conformance as the maximal conformance of all private views of a given public view.

**Definition 15 (Private view conformance).** Let \( N = N_1 \oplus \cdots \oplus N_k \) be a contract for \( X = \{X_1, \ldots, X_k\} \). Let \( N_i \) be the public view of \( X_i \), and let \( L_i \) be an event log of \( X_i \). Let \( Pr(N_i) = \{M \mid M \subseteq \text{acc } N_i\} \) denote the set of all private views that accord with \( N_i \). Then
Theorem 2 (Main result). Let $N = N_1 \oplus \cdots \oplus N_k$ be a contract for $X = \{X_1, \ldots, X_k\}$. Let $N_i$ be the public view of $X_i$, and let $L_i$ be an event log of $X_i$. Then $B_i = mpC(maxC(N_i))$ is a best matching private view for $N_i$ and $L_i$. 

Definition 16 (Fitness). Conformance $conf(L, N)$ w.r.t. to fitness of an event log $L$ and a labeled net $N$ yields a number between 0 (poor fitness) and 1 (perfect fitness) and is maximal if the alignment-based costs $\delta(L, N) = \sum_{\sigma \in L} \delta_N(\sigma)$ are minimal.

Our approach for deciding private view conformance does not rely on a specific fitness measure; any fitness measure is suitable as long as it meets the criteria in Def. 16. Our approach relies on the existence of two specific controllers of any open net $N$: a maximal controller $maxC(N)$ [14,7] and a most permissive controller $mpC(N)$ [22]. A maximal controller is maximal w.r.t. the accordance relation; that is, every controller of $N$ accords with $maxC(N)$. A most permissive controller $mpC(N)$ is maximal w.r.t. behavior; that is, $N$ can visit all the states in composition with $mpC(N)$ that can be visited in composition with any controller of $N$. For technical details of maximal and most permissive controllers we refer to [14] and [22], resp.; here, we only summarize their properties.

Proposition 1 ([14]). For any open net $N$, there exist controllers $maxC(N)$ and $mpC(N)$ such that for any controller $C$ of $N$, we have $C \sqsubseteq_{acc} maxC(N)$ and $Tr(env(C)) \subseteq Tr(env(mpC(N)))$.

Given a contract $N = N_1 \oplus \cdots \oplus N_k$, we show that $B_i = mpC(maxC(N_i))$ is a canonical best matching private view for $N_i$ and event log $L_i$. In other words, open net $B_i$ accords with $N_i$ and has minimal costs and, hence, maximal fitness.
Proof. Let \( N'_i \in \mathcal{P}(N_i) \) be a private view of \( N_i \). We prove \( \delta(L_i, env(N'_i)) \geq \delta(L_i, B_i) \), which implies \( conf(L_i, env(N'_i)) \leq conf(L_i, env(B_i)) \) for conformance w.r.t. fitness according to Def. 16. By the choice of \( N'_i \) and Prop. 1, we conclude \( R(env(N'_i)) \subseteq R(env(B_i)) \). Let \( \sigma \in L_i \) be a trace in the event log \( L_i \). Then \( I_{\sigma, \text{env}(N_i)} \subseteq I_{\sigma, \text{env}(B_i)} \) by Def. 11 and \( \delta(\lambda_{\text{env}(N'_i)}(\sigma)) \geq \delta(\lambda_{\text{env}(B_i)}(\sigma)) \) by Def. 12 and 13. Thus, \( \delta(L_i, N'_i) \geq \delta(L_i, B_i) \) by Def. 16.

Theorem 2 gives a theoretical solution for deciding private view conformance w.r.t. fitness. The question is, how can we actually calculate \( B_i \) for a given open net \( N_i \)?

In the next section, we show that \( B_i = mpC(maxC(N_i)) \) can actually be calculated, yet for acyclic open nets only. The reason for this restriction is that for acyclic open nets, the correctness notions weak termination and deadlock freedom coincide. The theory for maximal controllers in case of weak termination exists [7], but has not been implemented so far.

4 Experimental Results

Based on a prototypical implementation, we show first experimental results on computing a canonical best matching private view according to Thm. 2. We assume weak termination as a correctness criterion, use the asynchronous environment \( env^a \), and employ the standard distance function \( \delta_S \) to find the best matching alignments.

For the running example, \( \gamma_1 - \gamma_3 \) are best matching alignments for \( L \) and \( env(Public) \) with costs \( \delta_S(\gamma_1) = 0 \), \( \delta_S(\gamma_2) = 2 \), and \( \delta_S(\gamma_3) = 1 \), yielding alignment-based costs \( \delta(L, env(Public)) = 3 \). Likewise, \( \gamma_4 - \gamma_6 \) are best matching alignments for \( L \) and \( env(Private) \) with costs \( \delta_S(\gamma_4) = \delta_S(\gamma_5) = 0 \), and \( \delta_S(\gamma_6) = 1 \). Thus, \( \delta(L, env(Private)) = 1 \).

We compute the canonical best matching private view \( B \) of \( Public \) in three steps: (1) compute the maximal controller \( maxC(Public) \), (2) compute the most permissive controller \( B = mpC(maxC(Public)) \), and (3) calculate \( \delta(L, env(B)) \). Figure 4 shows the three steps and the tools involved. Our toolchain consists of a Bash script for deriving a best matching private view using Wendy [12], Maxis\(^1\), the PNapi [10], and ProM\(^2\). We illustrate our approach in the following.

---

\(^1\) http://svn.gna.org/viewcvs/service-tech/trunk/maxis/  
\(^2\) http://www.promtools.org/
Calculating $maxC(Public)$: Open net $maxC(Public)$ has 34 places and 45 transitions and was constructed following the approach presented in [14]: using the tool Wendy, we constructed an annotated automaton that represents all controllers of $Public$, derived the behavior of $maxC(Public)$ from it using the tool Maxis, and transformed the behavior into an open net using the PNapi. Figure 5 illustrates a part of $maxC(Public)$. As $maxC(Public)$ is a controller of $Public$, it has the same interface as $Public$ with input and output interchanged. Initially, this service fires nondeterministically one of the five transitions $tabd, \ldots, td$. Depending on the state reached, it can perform a number of sending or receiving events. For example, after firing $tabd$, the open net can receive $a$ or $b$ or send $d$.

Deriving $B$: In the second step, we calculate the most permissive controller $B = mpC(maxC(Public))$ of $maxC(Public)$. We constructed the behavior of $B$ using the tool Wendy and transformed it into open net $B$ using the PNapi. The resulting open net has 12 places and 22 transitions and is partly depicted in Fig. 6. $B$ and $Public$ are interface-equivalent. Consider the place $empty$. A token on $empty$ corresponds to a marking that is not reachable in the composition of $B$ and any controller of $Public$. As no controller of $Public$ initially sends a message $c$, transition $t8$ and $t9$ encode such “misbehavior” by producing a token on $empty$. When $empty$ contains a token and hence the composition will not be weakly terminating, every possible sending and receiving of messages is possible; thus, transitions $ta, tb, tc, td$ are connected to the correspondingly labeled interface places (indicated by the respective arcs without source or target). What we can see is that the behavior of $Private$ can be replayed on $B$. This shows, that it is not wrong to implement a specification such that the resulting implementation has more controllers than the specification. However, the added behavior cannot be used by any controller of the specification. In our example, no controller of $Public$ will initially send message $c$ although there exist implementations such as open net $Private$ that allow such behavior.

Checking conformance with $B$: By Thm. 2, $B$ is a best matching private view of $Public$. Therefore, in the last step, we calculate the alignment-based cost for $L$ and $env(B)$.
Fig. 6: The best matching private view $B = mpC(maxC(Public))$ of Public.

using the latest PNAlignmentAnalysis plug-in from the TU/e SVN repository\(^3\). We use the $A^*$-algorithm for cost-based fitness with default options. The best matching alignments for $L$ and $env(B)$ are

$$
\gamma_7 = \gamma_8 = \gamma_9 = \begin{bmatrix}
\tau & \alpha & d & \tau \\
\tau & \alpha & d & \tau \\
\tau & \alpha & d & \tau \\
\tau & \alpha & d & \tau \\
\tau & \alpha & d & \tau \\
\tau & \alpha & d & \tau \\
\tau & \alpha & d & \tau \\
\tau & \alpha & d & \tau \\
\end{bmatrix}
$$

with $\delta(\gamma_7) = \delta(\gamma_8) = \delta(\gamma_9) = 0$ yielding alignment-based costs $\delta(L, env(B)) = 0$. We see that $\delta(L, env(B))$ is indeed lower than $\delta(L, env(Public)) = 3$ and even lower than $\delta(L, env(Private)) = 1$.

We also generated five random public views and event logs using a modified version of the Process Log Generator\(^4\). This time, we used the synchronous environment $env^s$ for computing the private view conformance with ProM. All experiments were conducted on a MacBook Pro, Intel Core i5 CPU with 2.4 GHz and 8 GB of RAM. The results in Table 1 show that the average cost for each case (using the standard distance function) for conformance checking the log $L$ with the best matching private view $B$ (column 13) is significantly lower than conformance checking $L$ with the public view $N$ (column 11). This detail justifies Thm. 2. However, the lower cost come at a price of an exponentially larger size of $B$ (columns 1 and 5), which is caused by the construction of $B$ [14]. The net size results in a higher runtime of the $A^*$-algorithm (last row).

5 Related Work

Research on conformance checking of services follows two lines. One research line assumes a model of the implementation to be given (e.g., [20,5]) or that it is discovered from the event log (e.g., [15]). The former assumption is not always realistic. Furthermore, the result of conformance checking relies on the quality of the (discovered) model.

\(^3\) https://svn.win.tue.nl/repos/prom/

\(^4\) http://www.processmining.it/sw/plg
Table 1: Fully automatic private view conformance checking of synthetic nets.

<table>
<thead>
<tr>
<th>public view N</th>
<th>best matching B</th>
<th>event log L</th>
<th>cases</th>
<th>events</th>
<th>$\delta_S(N, L)$</th>
<th>time</th>
<th>$\delta_S(B, L)$</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>14 4 2 6</td>
<td>35 4 2 132</td>
<td>100 605</td>
<td>6.21</td>
<td>0.20</td>
<td>0.34</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16 5 3 8</td>
<td>41 5 3 190</td>
<td>100 541</td>
<td>7.53</td>
<td>0.20</td>
<td>0.88</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30 6 3 18</td>
<td>106 6 3 681</td>
<td>100 540</td>
<td>8.26</td>
<td>0.19</td>
<td>1.41</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>38 6 4 32</td>
<td>32 6 4 168</td>
<td>100 507</td>
<td>4.89</td>
<td>0.05</td>
<td>0.17</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>88 6 5 74</td>
<td>806 6 5 6,060</td>
<td>100 528</td>
<td>7.24</td>
<td>0.03</td>
<td>45.60</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The second research line assumes recorded behavior of the implementation to be given. Here, techniques are adapted from process mining [18,2]. Our contribution follows this research line. Van der Aalst et al. [4] map a contract specified in BPEL onto workflow nets (which can be seen as the synchronous environment) and employ conformance checking techniques from process mining [18]. In contrast, we measure the deviation of an implementation from its specification and all possible private views.

Comuzzi et al. [6] investigate online conformance checking using a weaker refinement notion than accordance. Different conformance relations on a concurrency-enabled model have been studied by De Leon et al. [8]. As their considered conformance relations differ from accordance, their work is not applicable in our setting (because maximal controllers have not been studied yet).

Motahari-Nezhad et al. [16] investigate event correlation; that is, they try to find relationships between events that belong to the same process execution instance. In contrast to event correlation, we do not vary the service instances, but refine the public view to a private view.

6 Conclusion

We presented an approach to calculate a best matching private view for a given event log and a public view. We proved the existence of a canonical best matching private view and showed that it can be automatically constructed—in the case of acyclic services and weak termination—using existing theory and tools on maximal controllers. Although it is possible to construct maximal controllers for cyclic services and weak termination [7], this has not been implemented yet.

A canonical best matching private view may become exponentially large in net size compared to its public view (see Table 1). It is an open question whether the current cost-based conformance checking techniques can be used for private view conformance checking for industrial service models. Another open question is how our approach can be generalized to deal with other quality dimensions (i.e., precision, generalization, simplicity), as in general there exist many best matching private views for a public view w.r.t. the fitness dimension.
References

Event Structures as a Foundation for Process Model Differencing, Part 1: Acyclic processes

Abel Armas-Cervantes, Luciano García-Bañuelos, and Marlon Dumas

Institute of Computer Science, University of Tartu, Estonia
{abel.armas, luciano.garcia, marlon.dumas}@ut.ee

Abstract. This paper considers the problem of comparing process models in terms of their behavior. Given two process models, the problem addressed is that of explaining their differences in terms of simple and intuitive statements. This model differencing operation is needed for example in the context of process consolidation, where analysts need to reconcile differences between process variants in order to produce consolidated process models. The paper presents an approach to acyclic process model differencing based on event structures. First the paper considers the use of prime event structures. It is found that the high level of node duplication inherent to prime event structures hinders the usefulness of the difference diagnostics that can be extracted thereon. Accordingly, the paper defines a method for producing (asymmetric) event structures with reduced duplication.

1 Introduction

Large companies with mature business process practices often manage multiple versions of the same process. Such variants may stem from distinct products, different types of customers (e.g. corporate vs. private customers), different legislations across countries in which a company operates, or idiosyncratic choices made by multiple business units over time. In some scenarios, analysts need to find ways to consolidate multiple process variants into a single one for the purpose of improving efficiency and creating economies of scale. In this setting, analysts need to accurately understand the differences between multiple variants in order to determine how to reconcile them. In this paper, we define a process model differencing operator that provides intuitive guidance to analysts, allowing them to understand the differences between a pair of process variants.

Behavioral profiles (BPs) [1] are a promising approach to tackle problems pertaining to the management of process variants. The idea behind BPs is to encode a process in terms of behavioral relations between every pair of tasks. In BPs, a pair of tasks is related by one of three types of relations:.strict order(\(\rightarrow\)), exclusive order(+ or interleaving(\(\|\|\)). BPs have been used for defining behavioral similarity metrics [2] and for process comparison and merging [3], among other applications. Nevertheless, BPs suffer from the following issues:

1. BPs do not correspond to any known notion of equivalence. Specifically, two processes may have the same BP while not being trace equivalent. Reciprocally, two processes may be trace equivalent, yet have different BPs.
2. **BPs mishandle the following patterns of behavior**: (a) task skipping, (b) behavior inside cycles – which is confused with interleaved parallelism – and (c) duplicate tasks.

Consider for example the variants of a land development process depicted in Figure 1(a)-(c), and their BPs presented aside. Firstly, even if we abstract away from task “c” both variants are non-trace equivalent, yet their BPs are identical (Issue 1). Secondly, note that task “b” is not always executed in the first variant, meaning that it can be skipped (Figure 1(a)). This fact is not captured in the BPs (Issue 2a). Finally, consider tasks “d”, “e” and “f” present in both variants. The order in which these tasks are executed is captured in both BPs using the *interleaving* relation. However, the actual order clearly differs between the two variants. This issue stems from the fact that these tasks are contained in a cycle and BPs confuse cycles and interleaved parallelism (Issue 2b).

![Variants of land development process](image)

Fig. 1. Variants of land development process

The paper makes a step forward by presenting an approach to process model differencing that solves issues 1 and 2a above for acyclic models – although it still suffers from not being able to handle cyclic models nor models with duplicate tasks. The presented approach is based on Event Structures (EVs), which allow us to ensure that two models are treated as equivalent iff they are fully concurrent bisimilar (cf. issue 1). First, the paper considers the use of Prime Event Structures (PES) for process model differencing. However, it is found that PESs inherently involve a significant amount of duplication, which degrades the usefulness of the diagnosis. To address this issue, the use of Asymmetric Event Structures (AES) [4] is considered. It is shown that AES can provide a more compact representation (less duplication), thus leading to a more compact diagnosis of differences.

In the proposed approach, differences between process models are captured by means of a (sparse) matrix wherein each cell represents a difference involving one or two tasks. This matrix can be directly translated into simple and intuitive statements. For instance, the difference between the process models depicted in
Figure 1 can be expressed in terms of statements of the following form:1 “In model (a), task b sometimes precedes \{d, e, f\}, whereas in model (b) task b always precedes \{d, e, f\}”, “Task c is present only in model (a)”, “In model (a), task d and e are executed in any order, while d always precedes e in model (b)”.

The discussion throughout the paper is developed assuming that the input process models are given as Petri nets.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 introduces some notions and notation used in the rest of the paper. Section 4 presents the acyclic process model differencing approach based on PES and AES. Finally, Section 5 concludes and discusses directions for future work.

2 Related work

Approaches for process model comparison may be divided into those based on node label similarity, process structure similarity and behavior similarity [5, 6]. In this paper we focus on behavior similarity. Nevertheless, we acknowledge that node label similarity plays an important in the alignment of nodes (e.g., tasks) across the process models being compared. Here, we assume that such an alignment is given, i.e. for each node label in one model we are given the corresponding (“equivalent”) node label in the other model. This is equivalent to assuming that the same node labels are used across both models being compared.

There exists a number of equivalence notions for concurrent systems [7]. Several methods for testing the equivalence of two systems according to these notions have been developed. However, most of these methods focus on determining whether or not two systems are equivalent. Relatively few previous research delves into the question of diagnosing all differences between two systems.

Perhaps one of the earliest work on diagnosing concurrent system differences is [8], which presents a technique to derive equations in a process algebra characterizing the differences between two Labeled Transition Systems (LTSs). This technique iteratively traverses the data structures used for testing bisimulation equivalence and generates a minimal equation characterizing differences for pairs of states in the input LTSs. However, the use of a process algebra makes the feedback difficult to grasp for end users (process analysts in our context). In [9], a method for assessing the dissimilarity of LTSs in terms of “edit” operations is presented. However, we contend that providing feedback in the form of a series of elementary edit operations (add or remove events) is not simple and intuitive, since it does not directly tell the analyst what relations exist in one model that do not exist in the other. [10] presents a method for diagnosing differences between pairs of process models using standard automata theory. Differences between a pair of models are captured by means of templates of the form “a node in a model has more dependencies than in the other one” or “a node in a model may be executed multiple times in one model but at most once in the other”. The method of [10] suffers from two major limitations. First, the set of reported

1 Note that the cycle is not taken into account in this comparison.
differences is not guaranteed to be complete. Second, the differences are given in a coarse-grained manner. With respect to the example in Figure 1, [10] gives as diagnosis that task “c” has additional dependencies in the first model viz. the second, without explaining the additional dependencies. We note that all three techniques above rely on LTSs and are not able to recognize concurrency relations. Thus, these techniques do not diagnose differences such as “in one model two tasks are done concurrently while in another they are done in sequence.”

In this paper, we rely on event structures to capture the behavior of process models. Event structures represent concurrent systems in terms of behavioral relations between events. Other representations of process models in terms of behavioral relations have been proposed in the literature. Causal footprints [11] represent behavior as a set of tuples, each comprising a task, its preset, and its postset. These sets of tuples are then mapped to points in a vector space and euclidian distance is used to compute a metric of similarity between a given pair of footprints. This technique however is not intended to provide a diagnosis of differences between pairs of models. Alpha relations are another example of a representation of process models using behavioral relations [12]. Alpha relations include direct causality, conflict and concurrency, and are derived from execution logs for the purpose of process mining. Alpha causality is not transitive. This choice makes alpha relations unsuitable for process model comparison, because causality has a localized scope. Moreover, alpha relations cannot capture so-called “short loops” and hidden tasks (including scenarios where a task may be skipped). The ordering relations graph [13–15] is another representation of (acyclic) process models in terms behavioral relations. However, it too has problems with hidden tasks. The class of asymmetric event structures characterized in the present paper is a refinement of the ordering relations graph.

3 Preliminaries

This section covers basic concepts on Petri nets and partially ordered sets. In particular, the notions of branching processes, behavior relations, and fully concurrent bisimulation are reviewed because they are cornerstones to our approach.

### 3.1 Petri nets

**Definition 1 (Petri net, Net system).** A tuple \((P, T, F)\) is a Petri net, where \(P\) is a set of places, \(T\) is a set of transitions, with \(P \cap T = \emptyset\), and \(F \subseteq (P \times T) \cup (T \times P)\) is a set of arcs. A marking \(M : P \rightarrow N_0\) is a function that associates each place \(p \in P\) with a natural number (viz., place tokens). A net system \(N = (P, T, F, M_0)\) is a Petri net \((P, T, F)\) with an initial marking \(M_0\).

Places and transitions are conjointly referred to as nodes. We write \(\bullet y = \{x \in P \cup T \mid (x, y) \in F\}\) and \(y \bullet = \{z \in P \cup T \mid (y, z) \in F\}\) to denote the preset and postset of node \(y\), respectively. \(F^+\) and \(F^\ast\) denote the irreflexive and reflexive transitive closure of \(F\), respectively.
The semantics of a net system can be defined in terms of markings. A marking $M$ enables a transition $t$ if $\forall p \in t : M(p) > 0$. Moreover, the occurrence of $t$ leads to a new marking $M'$, with $M'(p) = M(p) - 1$ if $p \in t \land t \bullet$, $M'(p) = M(p) + 1$ if $p \in t \bullet \land \neg t$, and $M'(p) = M(p)$ otherwise. We use $M \xrightarrow{t} M'$ to denote the occurrence of $t$. The marking $M_n$ is said to be reachable from $M$ if there exists a sequence of transitions $\sigma = t_1t_2\ldots t_n$ such that $M \xrightarrow{t_1} M_1 \xrightarrow{t_2} \ldots \xrightarrow{t_n} M_n$. A marking $M$ of a net is $n$-safe if $M(p) \leq n$ for every place $p$. A net system $N$ is said $n$-safe if all its reachable markings are $n$-safe. In the following we restrict ourselves to 1-safe net systems. Hence, we identify the marking $M$ with the set $\{ p \in P \mid M(p) = 1 \}$.

A labeled Petri net $N = (P,T,F,\lambda)$ is a net $(P,T,F)$, where $\lambda : P \cup T \to \lambda \cup \{ \tau \}$ is a function that associates a node with a label. Given a node $x$, if $\lambda(x) \neq \tau$ then $x$ is said to be observable, otherwise $x$ is said to be silent. A labeled net system $N = (P,T,F,M_0,\lambda)$ is similarly defined. An example of labeled net system is shown in Figure 2. There, transitions display their corresponding label inside the rectangle. Note that $t_2$ is a silent transition, hence $\lambda(t_2) = \tau$. Herein, we consider each place to be labeled with its corresponding identifier, e.g., $\lambda(p_0) = p_0$. The labeling of places is introduced for technical reasons and its usage will be clarified later on (cf., Definition 9).

### 3.2 Deterministic and branching processes

The partial order semantics of a net system is commonly formulated in terms of runs or, more precisely, prefixes of runs that are referred to as deterministic processes [16]. A process itself can be represented as an acyclic net with no branching nor merging places, i.e., $\forall p \in P : |\bullet p| \leq 1 \land |p\bullet| \leq 1$. Alternatively, all runs can be accommodated in a single tree-like structure, called branching process [17], that may contain branching places and explicitly represents three behavior relations: causality, concurrency and conflict defined as follows.

**Definition 2 (Behavior relations).** Let $N = (P,T,F)$ be a net and $x,y \in P \cup T$ two nodes in $N$.

- $y$ and $x$ are in causal relation, denoted as $x <_N y$, iff $(x,y) \in F^*$. The inverse causality relation is denoted $>_N$. We use $\leq_N$ to denote the reflexive causality relation.
- $x$ and $y$ are in conflict, denoted as $x \#_N y$, iff there exist two transitions $t,t' \in T$ such that $t$ and $t'$ are distinct, $t \cap t' = \emptyset$, and $(t,x),(t',y) \in F^*$. If $x \#_N x$ then $x$ is said to be in self-conflict.
- $x$ and $y$ are concurrent, denoted as $x \|_N y$, iff neither $x <_N y$, nor $y <_N x$, nor $x \#_N y$.

The tuple $\mathcal{R} = (<_N, \#_N, \|_N)$ denotes the behavior relations of $N$. We can now provide a formal definition for branching process.
Definition 3 (Branching process). Let $N = (P, T, F, M_0)$ be a net system. The branching process $\beta = (B, E, G, \rho)$ of $N$ is the net $(B, E, G)$ defined by the inductive rules in Figure 3. The rules also define the function $\rho : B \cup E \to P \cup T$ that maps each node in $\beta$ to a node in $N$. $\rho(B)$ is a shorthand for $\bigcup_{b \in B} \rho(b)$.

For sake of clarity, the elements of $B$ and $E$ in a branching process are called conditions and events, respectively. $\text{Min}(\beta)$ denotes the set of minimal elements of $B \cup E$ with respect to the transitive closure of $G$. Henceforth, $\text{Min}(\beta)$ corresponds to the set of places in the initial marking of $N$, i.e., $\rho(\text{Min}(\beta)) = M_0$. Figure 4 presents the branching process of the net system from Figure 2. Note that every node in the branching process has multiple labels. The label outside of the node, say “$e_0(t_0)$”, indicates that the underlying event is named “$e_0$” and that it corresponds to transition “$t_0$” in the net system. The events in the branching process also display the label in the original net system, e.g., “$d$”. This label can be determined by a composition of functions $\lambda$ and $\rho$, i.e., $\lambda_\beta \deq \lambda \circ \rho \circ \beta$.

One important characteristic of a branching process is that it does not contain merging conditions. To overcome this restriction, some nodes in the net system need to be represented more than once in the branching process. When comparing Figure 2 and Figure 4, we can notice that place $b_4$ and $b_5$, place $b_6$ corresponds to $b_6$ and $b_7$, and that transition $t_4$ corresponds to events $e_4$ and $e_5$.

Engelfriet [16] showed that every Petri net has a unique (possibly infinite) maximal branching process up to isomorphism, a.k.a. net unfolding. The branching process of an acyclic net system is finite.

We continue by formally defining deterministic processes, which will be used for introducing the notion of equivalence we rely on, namely fully concurrent bisimulation. By the same token, we introduce the notion of configuration.

Definition 4 (Configuration, deterministic process). Let $N = (P, T, F, M_0)$ be a net system and $\beta = (B, E, G, \rho)$ its corresponding branching process.

– A configuration $C$ of $\beta$ is a set of events, $C \subseteq E$, such that:
  i) $C$ is causally closed, i.e., $e \in C \Rightarrow e' \leq_\beta e; e' \in C$, and
  ii) $C$ is conflict free, i.e., $\forall e, e' \in C : \neg (e \#_\beta e')$.

– A deterministic process $\pi = (B_\pi, E_\pi, G_\pi, \rho)$ is the net induced by a configuration $C$, where $B_\pi = *C \cup *C$, $E_\pi = C$, and $G_\pi = G \cap (B_\pi \times E_\pi \cup E_\pi \times B_\pi)$.

– The initial deterministic process of $N$, denoted $\hat{\pi}$, is the process induced by the empty configuration $C = \emptyset$. 
Let $C$ and $C'$ be configurations of $\beta$, such that $C \subseteq C'$, and $\pi$ and $\pi'$ be the processes induced by $C$ and $C'$, respectively. Moreover, if $X = C' \setminus C$, then we write $\pi' = \pi \oplus X$ and we say that $\pi'$ is an extension of $\pi$. Given a process $\pi$ we are interested in extensions to $\pi$ appending exactly one observable event, whenever such extension exists, i.e., $\pi \oplus X$ such that $|\{x \mid x \in X \land \lambda_\beta(x) \neq \tau\}| = 1$. With abuse of notation, we shall write $\pi \oplus \lambda X$ to denote a process extension with exactly one observable event. In a branching process, each event $e$ can be associated with a unique set of events that causally precede $e$, denoted $[e] = \{e' \mid e' \preceq e\}$, and referred to as the local configuration of $e$.

### 3.3 Fully Concurrent Bisimulation

The notion of equivalence that we adopt in this work is known as fully concurrent bisimulation [18]. This bisimulation can be informally stated as follows: given two net systems, any sequence of observable events (viz., run or process) that might be possibly performed by one net system must also be possibly performed by the other net system and vice-versa, as for other conventional bisimulations, but additionally causal dependencies between observable events must be preserved in the bisimulation. The latter is required to be in-line with the partial order semantics. Below, we provide the corresponding formal definition.

**Definition 5 (Fully concurrent bisimulation).** Let $N_1 = (P_1, T_1, F_1, M_1^0, \lambda_1)$ and $N_2 = (P_2, T_2, F_2, M_2^0, \lambda_2)$ be labeled net systems, and $\beta_1 = (B_1, E_1, G_1, \rho_1)$ and $\beta_2 = (B_2, E_2, G_2, \rho_2)$ be their corresponding branching processes. The set of triples $B \subseteq \Pi_1 \times \Gamma \times \Pi_2$ is a fully concurrent bisimulation for $N_1$ and $N_2$ iff:

- $\Pi_1$ and $\Pi_2$ are the set of deterministic processes of $N_1$ and $N_2$, respectively, and $\Gamma : E_1 \to E_2$ is a relation from observable events in $\beta_1$ to observable events in $\beta_2$.
- If $\tilde{\pi}_1$ and $\tilde{\pi}_2$ are the initial deterministic processes of $N_1$ and $N_2$, respectively, then $(\tilde{\pi}_1, \emptyset, \tilde{\pi}_2) \in B$.
- $\Gamma$ is a bijection w.r.t. labeling, i.e., $\forall e \in \text{Dom}(\Gamma) : \lambda_{\beta_1}(e) = \lambda_{\beta_2}(\Gamma(e))$, preserving causality, i.e., $\forall e, e' \in \text{Dom}(\Gamma) : e \ll_{\beta_1} e' \Leftrightarrow \Gamma(e) \ll_{\beta_2} \Gamma(e')$.
- $\forall (\pi_1, \Gamma, \pi_2) \in B$:
  a) if $\pi'_1 = \pi_1 \oplus_{\lambda_1} X_1$ is an extension of $\pi_1$ with exactly one observable event, then there exists a tuple $(\pi'_1, \Gamma', \pi'_2) \in B$ with $\pi'_2 = \pi_2 \oplus_{\lambda_2} X_2$ and $\Gamma \subseteq \Gamma'$.
  b) if $\pi'_2 = \pi_2 \oplus_{\lambda_2} X_2$ is an extension of $\pi_2$ with exactly one observable event, then there exists a tuple $(\pi'_1, \Gamma', \pi'_2) \in B$ with $\pi'_1 = \pi_1 \oplus_{\lambda_1} X_1$ and $\Gamma \subseteq \Gamma'$.

Two net systems $N_1$ and $N_2$ are said fully concurrent bisimulation equivalent, denoted $N_1 \approx_{fc} N_2$, if there exists a fully concurrent bisimulation for them.

Note that fully concurrent bisimulation is defined in terms of process extensions with exactly one observable event and, hence, there is implicitly an abstraction of invisible behavior. This abstraction is convenient for our purposes, such that we lift it to the level of behavior relations. Let $N = (P, T, F, M_0, \lambda)$ be a labeled net system and $\beta = (B, E, G, \rho)$ be its branching process. Moreover, let $E'$ be the set of observable events, i.e., $E' = \{e \mid e \in E \land \lambda_\beta(e) = \tau\}$. Then
the following holds:

\[ \mathcal{R}_{\lambda} = (\prec, \cap E^2, \# \cap E^2, \| \cap E^2) \] defines the observable behavior relations, that is the behavior relations of \( N \) restricted to observable behavior.

If there exists a mapping of transitions in two net systems such that this mapping preserves their underlying behavior relations, then we say their behavior relations are isomorphic. This is formally defined as follows.

**Definition 6 (Isomorphism of observable behavior relations).** Let \( N_1 = (P_1, T_1, F_1, M_1^0, \lambda_1) \) and \( N_2 = (P_2, T_2, F_2, M_2^0, \lambda_2) \) be labeled net systems, and \( \beta_1 = (B_1, E_1, G_1, p_1) \) and \( \beta_2 = (B_2, E_2, G_2, p_2) \) be their corresponding branching processes. Moreover, let \( E_1' \subseteq E_1 \) and \( E_2' \subseteq E_2 \) be the set of observable events of the branching processes of \( N_1 \) and \( N_2 \), and \( \mathcal{R}_{\lambda_1} \) and \( \mathcal{R}_{\lambda_2} \) are their corresponding observable behavior relations. \( \mathcal{R}_{\lambda_1} \) and \( \mathcal{R}_{\lambda_2} \) are said isomorphic, denoted \( \mathcal{R}_{\lambda_1} \cong \mathcal{R}_{\lambda_2} \), if there exists a bijection \( \varphi : E_1' \rightarrow E_2' \) such that:

- \( \forall e \in E_1' : \lambda_{\beta_1}(e) = \lambda_{\beta_2}(\varphi(e)) \), and
- \( \forall e, e' \in E_1' : (e \prec_{\beta_1} e' \Leftrightarrow \varphi(e) \prec_{\beta_2} \varphi(e')) \lor (e \#_{\beta_1} e' \Leftrightarrow \varphi(e) \#_{\beta_2} \varphi(e')) \lor (e \|_{\beta_1} e' \Leftrightarrow \varphi(e) \|_{\beta_2} \varphi(e')) \).

The following Theorem establishes the relation between fully concurrent bisimulation equivalence and isomorphism of observable behavior relations for two net systems. The corresponding proof can be found in [13].

**Theorem 1 ([13]).** Let \( N_1 = (P_1, T_1, F_1, M_1^0, \lambda_1) \) and \( N_2 = (P_2, T_2, F_2, M_2^0, \lambda_2) \) be labeled net systems, and \( \beta_1 = (B_1, E_1, G_1, p_1) \) and \( \beta_2 = (B_2, E_2, G_2, p_2) \) be their corresponding branching processes with distinctive labelings. Moreover, let \( E_1' \subseteq E_1 \) and \( E_2' \subseteq E_2 \) be the set of observable events of \( \beta_1 \) and \( \beta_2 \), respectively. Assume there exists a bijection \( \psi : E_1' \rightarrow E_2' \) such that \( \forall e \in E_1' : \lambda_{\beta_1}(e) = \lambda_{\beta_2}(\psi(e)) \), then the following holds:

\[ N_1 \approx_{fcb} N_2 \iff \mathcal{R}_{\lambda_1} \cong \mathcal{R}_{\lambda_2} \]

### 3.4 Partial Ordered Sets

We conclude this section by recalling some definitions from the theory of partial ordered sets (posets) [17]. Let \( (D, \leq) \) be a poset. For a subset \( X \subseteq D \), an element \( y \in X \) is an upper (lower) bound of \( X \), if \( x \leq y \ (x \geq y) \), for each element \( x \in X \). An element \( y \in D \) is a greatest (least) element, iff for each element \( x \in D \) the property \( x \leq y \ (x \geq y) \) holds. An element \( y \in D \) is a maximal (minimal) element, if there is no element \( x \in D \) s.t. \( y \leq x \ (x \geq y) \); \( D_{\text{max}} \) and \( D_{\text{min}} \) denote the sets of maximal and minimal elements of \( D \), respectively. Two elements \( x, y \in D \) are consistent, denoted \( x \parallel y \), iff they have a joint upper bound, i.e., \( x \parallel y \Leftrightarrow \exists z \in D : x \leq z \land y \leq z \); otherwise, they are said inconsistent. A subset \( X \) of \( D \) is pairwise consistent, written \( X \parallel \), iff every pair of elements in \( X \) is consistent in \( D \); i.e., \( X \parallel \Leftrightarrow \forall x, y \in X : x \parallel y \). A poset \( (D, \leq) \) is coherent, iff each pairwise consistent subset \( X \) of \( D \) has a least upper bound (lub) \( \cup X \). An element \( x \in D \) is a complete prime, iff for each subset \( X \) of \( D \), with lub \( \cup X \), it holds \( x \leq \cup X \Rightarrow \exists y \in X : x \leq y \). We shall write \( \mathcal{P}_P \) to denote the set of complete primes of a poset \( P \). A poset \( P = (D, \leq) \) is prime algebraic, iff \( \mathcal{P}_P \) is denumerable.
and every element in $D$ is lub of the set of complete primes it dominates, i.e.,
\[ \forall x \in D : x = \cup \{ y | y \in P \land y \leq x \} \].

A set $S$ is denumerable, i.e., it is empty or there exists an enumeration of $S$ that is a surjective mapping from the set of positive integers onto $S$.

4 Comparison of acyclic process models

In this section, we explore the use of event structures (EVs) in the process model differencing. The presentation is organized in two parts. Subsection 4.1 introduces the more basic EV, namely Prime Event Structures (PESs). By the same token, we define an operator on event structures, that operationalizes the comparison of behavior relations. In spite of their faithful representation of behavior, PESs incur in a great amount of duplication. In Subsection 4.2, we consider Asymmetric Event Structures (AESs) as an alternative representation that reduces the amount of duplication observed in PESs.

4.1 Prime Event Structures

A Prime Event Structure (PES) is a model for computation introduced in [17].

Aligned with the concepts of in the previous section, we formally define PESs.

**Definition 7 ((Labeled) Prime Event Structure).** Let $N = (P, T, F, M_0, \lambda)$ be a net system and $\beta = (B, E, G, \rho)$ be its corresponding branching process. The prime event structure of $\beta$ is the tuple $\xi = (E, \preceq, \#)$, where $\preceq = \beta \cap E^2$ and $\# = \beta \cap E^2$. A labeled prime event structure also considers a labeling function $\lambda_\xi = \lambda_N \circ \rho$, that associates each event $e \in E$ with the label of its corresponding transition $t \in T$, i.e., $\lambda_\xi(e) = \lambda_N(t) \Rightarrow \rho(e) = t$.

The conflict relation $\#_\xi$ is said hereditary w.r.t. $\preceq_\xi$, meaning that for all $e, e', e'' \in E$, if $e \#_\xi e'$ and $e' \preceq_\xi e''$ then $e \#_\xi e''$. The behavior relations of a PES $\xi = (E, \preceq_\xi, \#_\xi)$ are given by the tuple $R_\xi = (\preceq_\xi, \#_\xi, E^2 \setminus (\preceq_\xi \cup \#_\xi \cup \#_\xi))$ [17]. Clearly, PES behavior relations correspond to the behavior relations of the corresponding branching process as introduced in Definition 2, restricted to the set of events. Armed with Theorem 1, we can restrict our focus to observable behavior. A PES without invisible behavior shall be denoted $\bar{\xi}$.

Figs. 5(a) and 5(b) present the PESs with and without invisible behavior for the net system in Fig. 2, as labeled graphs. There, nodes correspond to events, (solid) directed edges represent causality, e.g., $e_0 : d \rightarrow e_1 : e$ in Figure 5(b), whereas (dotted, decorated) undirected edges represent conflict, e.g., $e_3 : f \ldots \# \ldots e_4 : g$ also in Figure 5(b). Both transitive causal and hereditary conflict
relations are not shown to simplify the graph. When a pair of events is neither
direct nor transitively connected, such events are considered to be concurrent.
To further simplify the graphs, nodes in a PES will only display event labels and
not their identifiers, when it is clear from the context, e.g., Figures 5(c)–(d).

The PES $\xi_2$ is a variant of behavior for the net system in Figure 2, where the
transition labeled $f$ is not skipped (i.e., the silent transition $t_2$ is not present).
The differences in behavior of $\xi_1$ and $\xi_2$ are evident. There is one run in $\xi_1$
involving events $\{d,e,g\}$, cf., path highlighted with thick gray edges, and that is
not present in $\xi_2$. Note that the occurrence of event $e_4:g$ precludes that of event
$e_3:f$, because they are “in conflict”, which corresponds with the intuition that
the transition labeled $f$ may be skipped.

Alternatively, the behavior relations of PESs can be represented with matrices,
as illustrated in Figure 6. The subindexes of the relations were omitted for
the sake of readability. As usual, we write $R_{\xi}[e,e']$ to refer to the cell in the
intersection of the row associated to event $e$ and the column of $e'$. Therefore,
$R_{\xi}[e,f] = \parallel$ asserts the fact that events $e$ and $f$ are concurrent in $\xi_2$. Note that
all behavior relations are represented in the matrix, including the inverse causal
relation. Moreover, every event is self-concurrent, that by definition.

In order to characterize the differences on the behavior relations displayed
by two PESs, we define a binary operator as follows.

Definition 8 (Symmetric difference of PES behavior relations). Let
$\xi_1 = (E_1, \leq_{\xi_1}, \#_{\xi_1}, \lambda_{\xi_1})$ and $\xi_2 = (E_2, \leq_{\xi_2}, \#_{\xi_2}, \lambda_{\xi_2})$ be labeled prime event
structures, and let $R_{\xi_1} = (\leq_{\xi_1}, \#_{\xi_1}, \parallel_{\xi_1})$ and $R_{\xi_2} = (\leq_{\xi_2}, \#_{\xi_2}, \parallel_{\xi_2})$ be their cor-
responding behavior relations. The matching of events w.r.t. labelings $\mu \subseteq E_1 \times E_2$
and $\mu \subseteq E_1 \cup \{\alpha\} \times E_2 \cup \{\alpha\}$, where $\alpha$ is a marker for identifying missing events,
are defined as follows:
- $\mu = \{(e_1, e_2) | e_1 \in E_1 \land e_2 \in E_2 \land \lambda_{\xi_1}(e_1) = \lambda_{\xi_2}(e_2)\}$, and
- $\mu = \mu \cup \{(e_1, \alpha) | e_1 \in (E_1 \setminus \text{Dom}(\mu))\} \cup \{(\alpha, e_2) | e_2 \in (E_2 \setminus \text{Ran}(\mu))\}$.

Let $(e_1, e_2), (e'_1, e'_2) \in \mu$ be event matchings. The symmetric difference of $R_{\xi_1}$
and $R_{\xi_2}$, denoted $R_{\xi_1} \triangle R_{\xi_2}$, is defined as follows:

$$R_{\xi_1} \triangle R_{\xi_2} = \{(e_1, e_2), (e'_1, e'_2)\} =$$

$$= \begin{cases}
(R_{\xi_1}[e_1, e'_1], R_{\xi_2}[e_2, e'_2]) & \text{if } R_{\xi_1}[e_1, e'_1] = R_{\xi_2}[e_2, e'_2] \\
(\omega, R_{\xi_2}[e_2, e'_2]) & \text{if } R_{\xi_1}[e_1, e'_1] \neq R_{\xi_2}[e_2, e'_2] \\
(R_{\xi_1}[e_1, e'_1], \omega) & \text{if } e_1 = \alpha \lor e'_1 = \alpha \\
& \text{if } e_2 = \alpha \lor e'_2 = \alpha
\end{cases}$$

where $\omega$ stands for unspecified behavior relation.
Figures 7(a) presents the symmetric difference of the behavior relations of PES $\xi_1$, $\xi_2$ (cf., Figure 5). Every cell filled up with “$\cdot$” stands for a perfect match on the behavior of the process models being compared. Interestingly, in this example a large portion of behavior is matched. Let us consider the last column on the behavior of the process models being compared. Interestingly, in this scenario the number of events is exponential in size w.r.t. the maximal fan-out of the conflict places, i.e., $O(m^n)$ where $n$ is the number of events and $m$ is the average size of the poset of places. An additional side-effect is that the comparison of PESs requires a combinatorial number of matches among duplicate events.

\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline
 & (d,d) & (e,e) & (f,f) & (e_5,g,g) (e_4,g,g) & (d,d) & (e,e) & (f,o) (g,g) \\
\hline
(d,d) & - & - & - & - & - & - & (\#, <) \\
(e,e) & - & - & - & - & - & - & (\#, \|) \\
(f,f) & - & - & - & (\#, <) & - & - & (\#, \|) \\
(e_5,g,g) & - & - & - & (\#, \|) & - & - & (\#, \|) \\
(e_4,g,g) & - & - & (\#, >) & - & - & (\#, \|) & \\
\hline
\end{tabular}

(a) $R_{\xi_1} \triangle R_{\xi_2}$

(b) $R_{\xi_2} \triangle R_{\xi_1}$

\textbf{Fig. 7.} Example symmetric difference of the behavior relations of PESs in Figure 5

PESs provide a faithful representation of behavior but at the expense of duplication. This duplication stems from the fact that a new branch in the branching process is started at the point where each conflict place is found. Conversely, concurrency does not induce duplication. Hence, in the worst case scenario the number of events is exponential in size w.r.t. the maximal fan-out of the conflict places, i.e., $O(m^n)$ where $n$ is the number of events and $m$ is the average size of the poset of places. An additional side-effect is that the comparison of PESs requires a combinatorial number of matches among duplicate events.

4.2 Asymmetric Event Structures

To address the problems above, we consider an alternative to PESs known as Asymmetric Event Structures (AESs) [4]. In addition to the usual causality relation $\triangleright$, an AES introduces the asymmetric conflict relation $\triangleright \cdot$. Given two events $e, e' \in E$, we say that $e$ is in asymmetric conflict with $e'$, denoted $e \triangleright \cdot e'$, with...
two intuitive interpretations: (i) whenever both $e$ and $e'$ occur in a run, $e$ is observed before $e'$, and (ii) the occurrence of $e'$ precludes that of $e$. We note that in the original definition of AES, duplication of events is not a concern. Henceforth, we develop an approach to identifying the asymmetric conflict relation. We note that in [4] an AES is derived from a PES keeping the same number of events. Therefore, we develop an approach to identifying the asymmetric conflict relation while reducing the number of events.

When comparing the AES $\vartheta_1$ (cf., Figure 8(a)) with the PES $\xi_1$ (cf., Figure 5(b)), both of them displaying the observed behavior of the net system in Figure 2, one can immediately note a more compact representation. While in $\xi_1$, there are two events with the same label, namely $e_4:g$ and $e_5:g$, in $\vartheta_1$ only one event carries the label $g$. It turns out that the PES $\xi_2$ (cf., Figure 5(c)) is also an AES, such that we can compare the observable behavior relations $R_{\xi_1}$ and $R_{\vartheta_1}$ directly. The corresponding symmetric difference is presented in Figure 8(c).

In the following, we present a method to compute the asymmetric conflict relation. We start by computing an equivalence relation on the underlying branching process that relies on node labeling and that respects the local environment of events$^2$. Such an equivalence, referred to as future equivalence, has been introduced in [19] and can be formally stated as follows.

**Definition 9 (Future equivalence).** Let $\beta = (B, E, G, \rho, \lambda_\beta)$ be a labeled branching process. The relation $\sim \subseteq B^2 \cup E^2$ on nodes of $\beta$ is a future equivalence iff:

- $\forall x, x' \in B \cup E : x \sim x' \Rightarrow \lambda_\beta(x) = \lambda_\beta(x') \land (x \#_\beta x' \lor x = x')$, and
- $\forall e, e' \in E : e \sim e' \Rightarrow (\langle e \rangle)_- = (\langle e' \rangle)_- \land (\langle e \rangle)_+ = (\langle e' \rangle)_+.$

where $\langle x \rangle_- = \{ x' \mid x' \sim x \}$ and $\langle X \rangle_- = \{ \langle x \rangle_- \mid x \in X \}$.

Going back to the branching process in Figure 4, its corresponding future equivalence relation is $\{\{e_4:g, e_5:g\}, \{b_4:p_4, b_5:p_4\}, \{b_6:p_5, b_7:p_5\}\}$. Interestingly, the future equivalence relation can be used to fold a branching process into a Petri net that exhibits the same behavior (see [19, Theorem 8.7] for a formal proof). Indeed, after merging all future equivalent nodes of the branching process in Figure 4, we shall obtain the Petri net in Figure 2. An algorithm to compute future equivalences suitable for our setting is described in [20]. In contrast to previous work, we require future equivalent nodes to be in conflict: merging two concurrent events would eliminate one event from the run, merging two causal events would introduce a loop, both cases are undesirable in our setting.

Intuitively, an equivalence class $\langle e \rangle_-$ identifies a set of nodes in a branching process from which runs evolve isomorphically. Conversely, an equivalence class $\langle e \rangle_+$ can possibly have multiple different causes, collectively referred to as history. Such an intuition is formally defined as $\mathcal{H}(e) = \{\{e\}' \mid e' \in \langle e \rangle_+\}$. Now, let $e \in E$ be an event, then (i) the events in $\cap \mathcal{H}(e)$ are the strict causes of $e$, i.e., every event $e' \in \cap \mathcal{H}(e)$ is always observed before $e$, independently of the run, (ii) the events in $\cup \mathcal{H}(e) \setminus \cap \mathcal{H}(e)$ are the weak causes of $e$, i.e., every event $e' \in \cup \mathcal{H}(e) \setminus \cap \mathcal{H}(e)$ is observed before $e$ in at least one run, but not in all runs. The notion of weak

$^2$ The environment of an event is the set of conditions in its preset and postset.
causes of an event is indeed the way we use for identifying the asymmetric conflict relation. The following definition formalizes the transformation of an AES.

**Definition 10 (Branching process to AES).** Let $\beta = (B, E, G, \rho, \lambda_\beta)$ be a labeled branching process, and let $\sim$ be a future equivalence on $\beta$. The tuple $\vartheta = (E_\vartheta, \leq_{\vartheta}^{\text{full}}, \#_\vartheta, \rightarrow_\vartheta, \lambda_\vartheta)$ is the asymmetric event structure (AES) of $\beta$ induced by $\sim$, where

\[
E_\vartheta^{\text{full}} = \{|e| \mid e \in E\},
\leq_{\vartheta}^{\text{full}} = \{((e)_{\vartheta}, (e')_{\vartheta}) \mid e, e' \in E \land e \leq_\beta e'\},
\#_\vartheta = \{((e)_{\vartheta}, (e')_{\vartheta}) \mid e, e' \in E \land (e, e') \in \#_\beta \land \sim\},
\rightarrow_\vartheta = \{((e)_{\vartheta}, (e')_{\vartheta}) \mid e, e' \in E \land e \in \bigcup \mathcal{H}(e') \land \bigcap \mathcal{H}(e')\},
\lambda_\vartheta((e)_{\vartheta}) = \lambda_\beta(e), \text{ for all } e \in E.
\]

Note that the relation $\leq_{\vartheta}^{\text{full}}$ is a partial order and embeds the asymmetric conflict relation. Henceforth, for AESs we decompose $\leq_{\vartheta}^{\text{full}}$ into the strict cause relation, i.e., $<_\vartheta = <_{\vartheta}^{\text{full}} \setminus \rightarrow_\vartheta$, and the asymmetric conflict relation, i.e., $\rightarrow_\vartheta$. The relation $\leq_{\vartheta}^{\text{full}}$ can then be trivially derived from $<_\vartheta$ and $\rightarrow_\vartheta$. Technically, $e \#_\vartheta e'$ can also be represented as $e \rightarrow_\vartheta e' \land e' \rightarrow_\vartheta e$, but we consider that the symmetric conflict is more appropriate as feedback to modelers. As for PES, the concurrency relation on AES is given by $\parallel_\vartheta = E_\vartheta^{\text{full}} \setminus (<_{\vartheta}^{\text{full}} \cup \#_\vartheta \cup \rightarrow_\vartheta)$. The behavior relations of $\vartheta$ shall be denoted by the tuple $\mathcal{R}_\vartheta = (<_\vartheta, \#_\vartheta, \rightarrow_\vartheta, \parallel_\vartheta)$.

**Definition 11 (AES Configuration).** Let $\vartheta = (E_\vartheta, \leq_{\vartheta}^{\text{full}}, \#_\vartheta, \rightarrow_\vartheta, \lambda_\vartheta)$ be an AES. An AES configuration of $\vartheta$ is a set of events $X \subseteq E_\vartheta$ such that

- $<_\vartheta \cap C^2$ is well-founded,
- for all $e \in C$, if $e' <_{\vartheta}^{\text{full}} e \land e' \in C$, then there exists $e'' \in C$ s.t. $e' \#_\vartheta e''$.

As a consequence of the last condition, an AES configuration $C$ is conflict-free, i.e., $\forall e, e' \in C : \neg (e \#_\vartheta e')$. The set of all AES configurations of an AES $\vartheta$ shall be denoted $\text{Conf}(\vartheta)$. In spite of the differences in their definitions, the set of configurations of a branching process $\beta$ (cf., Definition 4) is the same as the set of AES configurations (cf., Definition 11) induced by a future equivalence $\sim$ on $\beta$. Intuitively, an equivalence class $(e)_{\vartheta}$ corresponds to a set of events in the branching process from which computation evolves independently, but isomorphically in different branches. When $(e)_{\vartheta}$ is mapped to a single event into an AES all the branches stemming from events in $(e)_{\vartheta}$ are merged into a single branch. However, the original set of configurations for $(e)_{\vartheta}$ can still be rebuilt by appending a copy of the merged branch to each configuration $[e']$ in $\mathcal{H}(e) = \{[e'] \mid e' \in (e)_{\vartheta}\}$. This intuition is formally confirmed in [20, Lemma 1]. Now, we define an order $\preceq$ on AES configurations, referred to as AES configuration extension, such that $C \subseteq C'$ stands for “$C$ can evolve into $C'$”.

**Definition 12 (AES Configuration Extension).** Let $\vartheta = (E_\vartheta, \leq_{\vartheta}^{\text{full}}, \#_\vartheta, \rightarrow_\vartheta, \lambda_\vartheta)$ be an AES and $X, X' \subseteq E_\vartheta$ sets of events. We say that $X'$ extends $X$, denoted $X \subseteq X'$, iff (i) $X \subseteq X'$, and (ii) $(e' <_{\vartheta}^{\text{full}} e)$ for all $e \in X, e' \in X' \setminus X$.

The relation above defines a partial order on the set of configurations of AES $\vartheta$, denoted $\mathcal{L}_\vartheta(\vartheta) = (\text{Conf}(\vartheta), \preceq)$. Following the approaches in [17] and [21], we now characterize $\mathcal{L}_\vartheta(\vartheta)$. 

Event Structures as a Foundation for Process Model Diﬀering 45
Proof. Let $X \subseteq Conf(\vartheta)$ be a set of pairwise consistent AES configurations, i.e., $X^\downarrow$. If $e,e' \in \cup\{X \mid X \in X\}$, then there exist $X,Y \in \mathcal{X}$ s.t. $e \in X$ and $e' \in Y$. Since $\mathcal{X}^\uparrow$ there exists an AES configuration $Z$ s.t. $X \subseteq Z$ and $Y \subseteq Z$. Therefore $\sim(e \neq a)$ since $Z$ is conflict-free, $\cup\{X \mid X \in \mathcal{X}\} \in Conf(\vartheta)$, and $\mathcal{X} = \cup \mathcal{X}$ is lub in $Conf(\vartheta)$. Hence, $\mathcal{L}_a(\vartheta)$ is coherent.

For any AES configuration $X \in Conf(\vartheta)$, we have $X = \cup\{[e]_X \mid e \in X\}$. Hence, if $X$ is complete prime, then there exists $e \in X$ s.t. $X = [e]_X$. This shows that the complete primes of $Conf(\vartheta)$ are AES configurations of the form $[e]_X$. Moreover, the formula $X = \cup\{[e]_X \mid e \in X\}$ alludes to the fact that any AES configuration is the lub of the complete primes it dominates.

Given the poset $\mathcal{L}_a(\vartheta) = (Conf(\vartheta), \subseteq)$, we know from [17] that $\mathcal{P}(\mathcal{L}_a(\vartheta)) = (\mathcal{P}_P, \subseteq, \#)$ is a prime event structure, where $\subseteq$ is $\subseteq$ restricted to $\mathcal{P}(\mathcal{L}_a(\vartheta))$, and for all $[e],[e'] \in \mathcal{P}(\mathcal{L}_a(\vartheta)) : [e] \neq [e']$ iff $[e]$ and $[e']$ are inconsistent in $\mathcal{L}_a(\vartheta)$. Moreover, a labeling function for such PES is given by $\lambda_{\mathcal{P}(\mathcal{L}_a(\vartheta))}([e]) = \lambda_{\vartheta}(e)$ for all $[e] \in \mathcal{P}(\mathcal{L}_a(\vartheta))$. By applying the notions above, we can derive the poset shown in Figure 9(a) and then the PES in Figure 9(b). The isomorphism of Figures 9(b) and 5(a) is not a surprise, considering that they correspond to isomorphic posets [17, Theorem 9]. All transformations employed in the approach are summarized in Figure 9(c).

An AES built using Definition 10 considers all the equivalence classes in the branching process. However, the comparison of AESs may happen in two phases, first with observable behavior and then with a subset of $\tau$-labeled events. We conjecture that only a subset of $\tau$-labeled events needs to be kept to allow the construction of AES configurations and preserve fully concurrent bisimulation.

5 Conclusion

In this work, we defined an acyclic process model differencing operator based on prime event structures (PESs) and asymmetric event structures (AESs). The
high level of duplication inherent to PES hinders on the usefulness of this representation for the purpose at hand. Hence, AESs were considered as an alternative to reduce duplication. A tailor-made method for computing AESs was described.

We foresee a number of avenues for future research. First, we aim at characterizing the minimal set of $\tau$-labeled events required to preserve fully concurrent bisimulation. Naturally, we target to extend the current approach so as to cover process models with cycles and duplicate tasks. A promising direction for dealing with cyclic process models includes techniques of net unfoldings for finding a finite representation of cyclic behavior. Finally, future research will include experiments to assess the readability of the feedback stemming from this approach and its applicability in real-world settings.

Acknowledgments. This research is partly funded by ERDF via the Estonian Center of Excellence in Computer Science and the Estonian Science Foundation.

References

Formal Modeling and Analysis of the REST Architecture Using CSP

Xi Wu, Yue Zhang, Huibiao Zhu, Yongxin Zhao, Zailiang Sun and Peng Liu

Shanghai Key Laboratory of Trustworthy Computing
Software Engineering Institute, East China Normal University
3663 Zhongshan Road (North), Shanghai, China, 200062
{xiwu,yzhang,hbzhu,yxzhao,zlsun,liup}@sei.ecnu.edu.cn

Abstract. As one of the most promising architectural styles, Representational State Transfer (REST) was first proposed to support the enablement of scalable and reliable design for largescale distributed hypermedia systems such as the World Wide Web (WWW). Rapidly development of the RESTful systems brings the misunderstanding and misapplied of the REST architecture. In this paper, we present a formal model to capture the essential features for the REST architecture, in which components in RESTful systems are modeled as CSP processes. Thus all the REST constraints can be completely described and validated in our framework. Furthermore, many REST constraints are verified using the model checker PAT. The proposed framework for REST architecture is not only confined to HTTP but can also be applied to other REST-compliant protocols. Finally a case study about an application scenario for environment monitoring is illustrated to show the feasibility of our approach. Consequently, better understanding of REST can be achieved and implementations of RESTful systems can benefit from it.

1 Introduction

REST architectural style, which was proposed for distributed hypermedia systems by Fielding in 2000 [6], has been widely used in practice [1, 8]. It is derived from several web-based architectural styles [14, 18] with a number of additional constraints, such as the Client-Server constraint and the Uniform Interface constraint, etc. Nowadays, REST architectural style being more and more widespread in real applications and it has already affected the design of many web-based applications [7, 13], especially for the Mashup [2, 4] and the World Wide Web (WWW) [3]. Unfortunately, the increasing interest of REST also brings the misunderstanding and misapplied of the REST architecture. Therefore, giving a good understanding of REST architectural style has been becoming more and more important for the guidance of the development of web-based applications, which is also the main goal of this paper.

Systems are often referred to as RESTful systems if they cater to the REST constraints. In the literature, as far as we know, there is little on formal modeling and analysis of RESTful system. In [11], Koch has already established a formal model for
hypermedia systems, but the model may cover many unRESTful properties about the
WWW instead of the fundamental principles of REST. Zuzak et al. [21] have proposed
a framework, which is based on a nondeterministic finite-state machine with epsilon
transitions for modeling the RESTful systems and they have also explained how to map
some REST principles to the model. They have done well and they hope to add some
other principles into their model, including the layered and cacheable style constraints
which are currently unaddressed in their model. In addition, Autonio et al. [8] have
given a precise definition of RESTful semantic web services combining tuple space
computing and process calculi, and they focus more on the semantics and application
of REST instead of the architecture itself. Moreover, the model in their paper may be
limited to standard HTTP methods and the other REST-compliant protocols are not
mentioned in their paper. In [10], Uri Klein et al. have formulated two key REST prop-
erties and RESTful HTTP within temporal logic. However, the other constraints and the
other REST-compliant protocols have not been mentioned in their formalization. Thus,
the research for formalizing the RESTful systems is still challenging.

Inspired by Fielding’s and Zuzak’s works [6, 21], in this paper, we use CSP [9] to
model and analyze the REST architecture and map its style constraints to the model.
CSP is a well-known process algebra in modeling and reasoning about the web ser-
vices [20]. We abstract the REST architecture and all of its components, including User
Agent, Origin Server, Intermediary Component and Cache, are described as CSP pro-
cesses, thus the communication between each entity is converted to the communication
between CSP processes. We also map the REST constraints to our model, for instance,
each component is described as an independent process, so that the Client-Server and
the layered structures are obvious. More details about the mapping of the six constraints
to our model can be found in our paper. Our framework is not limited to standard HTTP
but also can be applied to other REST-compliant protocols such as CoAP (Constrained
Application Protocol). Besides, we also use PAT, a model checker tool based on CSP, to
verify the Client-Server and the Cacheable constraints in our achieved model to prove
that our model caters to the REST architecture. Finally, giving a case study, we use
our framework to model and reason about an application scenario used for sensor data
retrieval adopting CoAP in order to show the feasibility of our model. The main contri-
butions of this paper are listed as follows:

• Modeling. A formalization of the REST architecture (user agent, intermediary
components, origin server and cache), based on the process algebra CSP, is pre-
sented. An abstract process algebra model for each REST architecture element is
given and six constraints characterizing REST are considered.

• Analysis. We analyze the constraint features of REST and explain the mapping
of the six constraints to the achieved model. Besides, PAT, the model checker, is
applied in verifying some constraints in our model to prove that our model caters
to the REST architecture.

• Application. A case study about environment monitoring is given to show the fea-
sibility of our model. It also illustrate that our framework is not confined to HTTP
but also can be applied to other REST-compliant protocols.

The remainder of this paper is organised as follows. We introduce preliminaries
about CSP and PAT in Section 2. Section 3 is devoted to the introduction of the REST
architecture including its elements and six constraints. We propose a formalization framework in Section 4 and all the communication components are described as CSP processes. In Section 5, we revisit six REST constraints and show how we map them to the achieved model. In this section, we also use the model checker tool PAT to verify the constraints. Giving a case study, we apply our framework in modeling and reasoning about an application scenario in Section 6. We conclude the paper and present the future directions in Section 7.

2 Preliminaries

2.1 CSP Method

Communicating Sequential Processes (abbreviated as CSP) was proposed by C. A. R. Hoare in 1978 [9]. It has developed and evolved constantly, and it has already become a mature process algebra. It specializes in describing the interaction between concurrent systems using mathematical theories. Due to powerful expressive ability, CSP is widely applied in many fields [15, 16, 19] such as the web service. In CSP, processes are composed of basic processes and actions, and they are connected by operators.

The syntax of CSP is shown below:

\[ P, Q ::= \text{Skip} | \text{Stop} | a \rightarrow P | P ; Q | c?x \rightarrow P | c!x \rightarrow P | P \parallel Q | P | X | P < b \triangleright Q \]

Here \( P \) and \( Q \) represent processes which have alphabets \( \alpha(P) \) and \( \alpha(Q) \) denoting the actions that the processes can perform respectively. While \( a \) and \( b \) stand for the atomic actions and \( c \) is the name of the channel.

- \( \text{Skip} \) represents a process which does nothing but terminates successfully.
- \( \text{Stop} \) denotes that the process is in the state of deadlock and does nothing.
- \( a \rightarrow P \) represents that the process first performs action \( a \), then behaves the same as process \( P \).
- \( P ; Q \) means performing \( P \) and \( Q \) sequentially.
- \( c?x \rightarrow P \) gets a message through channel \( c \) and assigns it to a variable \( x \), then behaves like \( P \).
- \( c!x \rightarrow P \) sends a message \( x \) using channel \( c \), then behaves like \( P \).
- \( P \parallel Q \) describes the concurrent between \( P \) and \( Q \).
- \( P | X | Q \) represents that \( P \) and \( Q \) perform the concurrent events on set \( X \) of channels.
- \( P < b \triangleright Q \) means if the condition \( b \) is true, the behavior is like \( P \), otherwise, like \( Q \).

The above is a brief description of CSP. More details can be found in [9].

2.2 PAT

In this subsection, we give a brief introduction to PAT [17]. PAT (Process Analysis Toolkit) is designed as an extensible and modularized framework based on CSP and it has been used to model and verify a lot of different systems for varieties of properties.
such as deadlock-freeness, divergence-freeness, reachability, LTL properties with fairness assumptions, refinement checking and probabilistic model checking. We list some notations in PAT as follows:

1. \#define \textit{N} 0 defines a global constant \textit{N} which has the initial value 0.
2. \texttt{var cache} \_\texttt{user} \_\texttt{list}[\textit{N}] defines an array named \texttt{cache}\_\texttt{user}\_\texttt{list} and the size of it is \textit{N}.
3. Channel \texttt{c} \_\texttt{5} defines a communication channel and the capacity of it is 5.
4. \texttt{P} = \{ \texttt{x} = \texttt{x} + 1 \} \rightarrow \texttt{Skip} defines an event that can be attached with an assignment, using which we can update the value of a global variable \textit{x}.
5. \texttt{c!a.b \rightarrow P} and \texttt{c?x.y \rightarrow P} refer to sending message \textit{a.b} and receiving message from channel \textit{c} respectively.

Besides, PAT can also describe the control flow structures, such as \texttt{if} -- \texttt{then} -- \texttt{else} and \texttt{while}, etc. More details about this tool can be found in [5, 12].

3 Overview of the REST Architecture

Representational State Transfer (REST) is a software architectural style designed for distributed hypermedia systems. It was first presented by Roy Fielding in his doctoral dissertation in 2000 [6]. World Wide Web (WWW) [3] is the largest implementation conforming to the REST architectural style. REST makes it clear that a web architecture can be obtained by characterizing and constraining the micro-interactions of the components of the architecture.

3.1 The Elements of REST

The REST architectural elements include components, connectors and data elements. Data elements are made up of resources, resource identifiers, representations, representation metadata, resource metadata and control data. Connectors are interfaces which components are used to communicate. They are made up of clients, servers,

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig1.png}
\caption{The framework of the REST architectural style}
\end{figure}
caches, resolvers and tunnels. Connectors encapsulate the underlying implementation of resources and the communication mechanisms. Components include user agents, origin servers and intermediary components (proxies and gateways). In order to illustrate the architecture of REST clearly, we give a simple framework in Figure 1.

REST identifies the resources involved in the interactions of different components with the use of resource identifiers. Connectors serve as uniform interfaces to access the set of resources. User agents use client connectors to send requests for resources and become the final receiver of corresponding responses. Origin servers use server connectors to manage requested resource namespace. The final receiver of a request to modify the values of resources should be an origin server. The intermediary components include proxies and gateways. Proxies are selected by clients to provide interface encapsulation for data translation or security protection. Gateways are selected by network or an origin server to provide interface encapsulations for other services such as data translation and security enforcement.

3.2 The Constraints of REST

The REST architectural style describes six constraints, including the Client-Server constraint, the Stateless constraint, the Cacheable constraint, the Layered constraint, the Uniform Interface constraint and the Code-On-Demand constraint. A system is usually referred to as a RESTful system if it conforms to these constraints.

1. **Client-Server**: Servers are only responsible for data storage and a uniform interface separates clients from servers. Clients and servers can evolve independently as long as the interfaces between them remaining unchanged.

2. **Stateless**: Servers are stateless and clients store session states. Each request contains all the information that receivers need to process it.

3. **Cacheable**: Clients can determine to or not to store cacheable responses by restricting responses from servers with a property indicating whether they are cacheable or not.

4. **Layered**: Clients originally don’t have any knowledge about the components which directly connecting to are end servers or intermediaries in a complicated system. Intermediaries improve the scalability of the system, providing shared cache and load-balance mechanisms. They can also enforce the security of the system.

5. **Uniform Interface**: The uniform interfaces between clients and servers simplify the architecture and enable clients and servers to evolve independently.

6. **Code-On-Demand**: This is an optional constraint. Servers can send executable codes to clients for the expansion of the functions of clients.

REST specifies four principles to standardize the implementation of uniform interfaces. The first one is the identification of resources. Web-based RESTful systems identify resources using URIs. The second one is the manipulation of resources through representations. Clients can modify or delete resources on servers by manipulating the representations of resources. The third one is the self-description of messages. Each message contains enough information describing how to handle the message. The last one is hypermedia as the engine of application state. Clients make state transitions only through actions that are dynamically identified within hypermedia by the server.
4 Formalization

In this section, we apply CSP in modeling the REST architectural style, including the components of the user agent, the cache, the origin server and the intermediary components (proxies and gateways). Firstly, we define the sets and the channels below.

We define the set \textbf{User} of user agents, \textbf{Server} of origin servers, \textbf{Cache} of caches and \textbf{Intermediary} of intermediary components (proxies and gateways). The set \textbf{Resource} contains the resources which are required by the user agent and \textbf{Url} indicates the identification of resources. Besides, we also define the set \textbf{Representation} of representations of the resource, \textbf{SDInformation} of the self-descriptive messages which can describe the message itself using its own data and metadata, and \textbf{HyperMedia} of the hyper media of the resources. In addition, we also define the messages which are delivered among the components as follows:

\[
\text{MSG}_{\text{req}} = \{ \text{msg}_{\text{req}}, \text{info}_{\text{cons}}, \text{sender}, \text{receiver} \mid \\
\text{info}_{\text{cons}} = (\text{url}, \text{repr}, \text{sdi}, \text{hmedia}), \text{url} \in \text{Url}, \\
\text{repr} \in \text{Representation}, \text{sdi} \in \text{SDInformation}, \\
\text{hmedia} \in \text{HyperMedia}, \text{sender} \in \text{User} \cup \text{Intermediary}, \\
\text{receiver} \in \text{Cache} \cup \text{Intermediary} \cup \text{Server} \}
\]

\[
\text{MSG}_{\text{rep}} = \{ \text{msg}_{\text{rep}}, \text{content}, \text{sender}, \text{receiver} \mid \\
\text{content} = (\text{url}, \text{repr}, \text{sdi}, \text{hmedia}), \text{url} \in \text{Url}, \\
\text{repr} \in \text{Representation}, \text{sdi} \in \text{SDInformation}, \\
\text{hmedia} \in \text{HyperMedia}, \text{sender} \in \text{Cache} \cup \text{Intermediary} \cup \text{Server}, \\
\text{receiver} \in \text{User} \cup \text{Intermediary}, \text{content} \in \text{Resource} \}
\]

\[
\text{MSG} = \text{MSG}_{\text{req}} \cup \text{MSG}_{\text{rep}}
\]

Here, $\text{MSG}_{\text{req}}$ represents the set of the request messages which are sent by the user agent and the intermediary component. The user agent can send request to its cache or the intermediary component, and the intermediary component can also send request messages to its own cache or the origin server. $\text{MSG}_{\text{rep}}$ denotes the reply messages that are sent back to the user agent or the intermediary component. The reply messages may be from the server or caches. Note that $\text{info}_{\text{cons}}$ and $\text{content}$ are four-tuples and $\text{content}$ is the resource entity returned to the user agent.

We use four channels to model the communication between each component: \textbf{ComUC}, \textbf{ComCIC}, \textbf{ComUIC}, \textbf{ComICS}.

- \textbf{ComUC}: it is used to send and receive messages between the user agent and its cache.
- \textbf{ComCIC}: it denotes the standard communication between the intermediary component and its cache.
- \textbf{ComUIC}: the user agent uses it to send request messages to the intermediary component and also receive the reply messages.
- \textbf{ComICS}: the intermediary component communicate with the origin server through this channel.

The declaration of the channels are as follows:

\textbf{Channel} \textbf{ComUC}, \textbf{ComCIC}, \textbf{ComUIC}, \textbf{ComICS}: \text{MSG}

Figure 2 illustrates the communications among components using channels.
4.1 User Agent

In the whole process of web service, when the user agent is prepared to send a request message, it checks its local cache for cacheable responses it needs first. That is to say, the user agent makes a request to its cache at first. If there are already cacheable responses in its local cache, the user agent can get the resource directly from the cache. Otherwise, the user agent will make requests to the outer network. The user agent can send the request message to the intermediary component for further processing. We model the behaviors of the user agent as follows:

\[ UserAgent(info\_cons) = df \]
\[ ComUC!request.info\_cons.U.C \rightarrow ComUC?reply.content.C.U \rightarrow \]
\[ UserAgent(info\_cons) < (content! = NULL) \geq \]
\[ ComUIC!request.info\_cons.U.IC \rightarrow \]
\[ ComUIC?reply.content.IC.U \rightarrow \]
\[ UserAgent(info\_cons) \]

Note that, the parameter info\_cons is a four-tuple \((url, repr, sdi, hmedia)\). All of the elements in the tuple are used to describe the resource. url stands for the identification of the resource and repr is the representation. sdi is the self-descriptive message of the resource and hmedia is hypermedia. The parameter info\_cons is a variable, which will change with the changing of these four elements.

4.2 Cache

Here, caches are used to store early received cacheable responses from servers. User agent can quickly access desired data in caches before sending request messages to the origin server, so caches can reduce the number of requests sent to servers and improve the efficiency of the whole system. If there are received cacheable responses, the cache will send the content of the resource back to the user agent. Otherwise, it will send a NULL message to its user. The communication between the intermediary component and its cache is the same as the process between the user agent and its cache. We model
the behaviors of the cache below:

\[
\text{Cache}(U, \text{info}_{\text{cons}}) =_{df} \text{ComUC?request.info}_{\text{cons}.U,C} \rightarrow \\
\text{ComUC!reply.content.C.U} \rightarrow \text{Cache}(U, \text{info}_{\text{cons}})
\]

\[
\text{Cache}(IC, \text{info}_{\text{cons}}) =_{df} \text{ComCIC?request.info}_{\text{cons}.IC,C} \rightarrow \\
\text{ComCIC!reply.content.C.IC} \rightarrow \text{Cache}(IC, \text{info}_{\text{cons}})
\]

\[
\text{Cache}(\text{Node}, \text{info}_{\text{cons}}) =_{df} \text{Cache}(U, \text{info}_{\text{cons}}) \langle \text{(Node} == U) \rangle \langle IC, \text{info}_{\text{cons}} \rangle
\]

Note that, \(\text{Cache}(\text{Node}, \text{info}_{\text{cons}})\) will perform different processes according to the value of \(\text{Node}\).

### 4.3 Intermediary Component

The intermediary component here plays an important role in the REST architectural style. The user agent will send the request message to the intermediary component if it cannot get the resource directly from its cache. The intermediary component also has cache which will be accessed first after the component receives the request message from the user agent. The behaviors of the intermediary component have two types, one is communicating with the user agent and the other one is communicating with the origin server. Here, we use CSP to model the behaviors of the intermediary component as follows:

\[
\text{Intermediary(info}_{\text{cons}}) =_{df} \\
\text{ComUIC?request.info}_{\text{cons}.U,IC} \rightarrow \\
\text{ComCIC!reply.content.C.IC} \rightarrow \\
(\text{ComUIC!reply.content.C.U} \rightarrow \text{Intermediary(info}_{\text{cons}})) \\
\langle (\text{content}! = \text{NULL}) \rangle \langle \text{ComICS!request.info}_{\text{cons}.IC,S} \rightarrow \\
\text{ComICS!reply.content.S.IC} \rightarrow \text{ComUIC!reply.info}_{\text{cons}.IC.U} \rightarrow \\
\text{IntermediaryComponent(info}_{\text{cons}})\rangle
\]

### 4.4 Origin Server

In the whole process of web service, the resources that the user agent needs are stored in the origin server. If there is no early received cacheable responses from servers stored in the user agent’s cache, the user agent will communicate with the server through the intermediary component. Here, the server may receive the request message from the intermediary component and also return the resource to it. We model the behaviors of the origin server below:

\[
\text{Server(info}_{\text{cons}}) =_{df} \\
\text{ComICS?request.info}_{\text{cons}.IC,S} \rightarrow \\
\text{ComICS!reply.content.S.IC} \rightarrow \text{Server(info}_{\text{cons}})
\]
4.5 System

The whole system can be modeled as a concurrent composition of the user agent, the cache, the intermediary component and the server.

\[\text{USERAGENT} = _{df} \text{UserAgent}(\text{info, cons}) || \text{ComUC} || \text{Cache(U, info, cons)}\]

\[\text{INTERMEDIARY} = _{df} \text{Intermediary}(\text{info, cons}) || \text{ComCIC} || \text{Cache(IC, info, cons)}\]

\[\text{SERVER} = _{df} \text{Server}(\text{info, cons})\]

\[\text{SYSTEM} = _{df} \text{USERAGENT} || \text{ComUIC} || \text{INTERMEDIARY} || \text{ComICS} || \text{SERVER}\]

5 REST Constraints Revisited

In this section, we mainly give a detailed description about how we map the six REST constraints to our achieved model. Based on the traces analysis, we give the formal definitions of all the constraints, and we use a model checker tool PAT, which we have already introduced in Section 2, to verify some constraints to prove that our achieved model caters to the REST architecture. First we define some functions below:

- \(\text{source}(\text{event})\) returns the sender of the event \(\text{event}\) and \(\text{source}(\text{event}) \in \text{Sender}\).
- \(\text{target}(\text{event})\) returns the receiver of the event \(\text{event}\) and \(\text{target}(\text{event}) \in \text{Receiver}\).
- \(\text{type}(\text{event})\) stands for the type of the event \(\text{event}\) and \(\text{type}(\text{event}) \in \text{Type}\).
- \(\text{set}(\text{trace})\) returns all the elements of the trace \(\text{trace}\).
- \(\text{trace}(\text{system})\) returns the trace of the \(\text{system}\).
- \(\text{channel}(\text{event})\) returns the channel name, on which the processes send and receive events. Thus \(\text{channel}(\text{event}) \in \text{Channel}\).

Here, we give the definitions of some additional sets. \text{Sender} is a set containing all the senders that can send an event. The type of all the elements in the set is enumerated, such as \{client, cache, server, intermediary\}, the same as the set \text{Receiver}. The set \text{Type} stands for the type of an event, elements in which are also enumerated, such as \{request, reply\}. The set \text{Channel} contains all the channels.

5.1 Constraints Descriptions

In this subsection, before we give the formal descriptions of six REST constraints, some notations are defined below.

1. \(\bullet\) represents \text{satisfy}, e.g. \(a \bullet A\) means that \(a\) satisfies \(A\).
2. // stands for \text{after}, e.g. \(\text{system/}\text{tr}\) means the process \text{system} has done the trace \text{tr}. Thus \(\text{trace(}\text{system/}\text{tr})\) represents the trace after the process \text{system} has done the trace \text{tr}.
3. \(\overline{\text{tr}}\) is a new trace which is the reversion of trace \text{tr} and \(\text{tr}_0\) stands for the first event of trace \text{tr}. Thus, \(\overline{\text{tr}}_0\) represents the first event of the reversion of the trace \text{tr}. \text{tr}' means the remaining of trace \text{tr} except the first event.
Client-Server

In our model, we have the process UserAgent to stand for the client connector and the process OriginServer for the server connector. Being as two independent processes, UserAgent and OriginServer are separated through the uniform interface, which is the channel in this paper. Request messages will be sent to the cache or the origin server if the user agent is intended to get resources. Based on the traces analysis, the Client-Server constraint can be formally defined as follows:

**Constraint 1:**

$$\forall tr \in \text{trace(system)}, \exists e \in \text{set(trace(system/tr))} \bullet \left( \text{source}(tr_0) == \text{client} \land \text{type}(tr_0) == \text{request} \right) \Rightarrow \left( \text{source}(e) == \text{cache} \lor \text{source}(e) == \text{server} \land \text{type}(e) == \text{reply} \right)$$

The resource the user agent is intended to get is in its cache or in the origin server. Whenever the user agent sends a request message, it will receive a reply which is from its cache or from the origin server.

Cacheable

In our model, the user agent and the intermediary component have caches, which are abstracted as the CSP process Cache. If a response in the reply is cacheable, it will be stored in the cache and reused next time. Note that there is a special mark in the response to show whether the response is cacheable or not. Here, we ignore the responses which can not be cached, that is we only consider the cacheable responses. The formal description is listed below:

**Constraint 2:**

$$\forall tr \in \text{trace(system)}, \forall e \in \text{set}(tr), \exists e' \in \text{set(trace(system/tr)))} \bullet \left( \text{source}(tr_0) == \text{client} \land \text{type}(tr_0) == \text{request} \land \text{source}(e) == \text{client} \land \text{type}(e) == \text{request} \right) \Rightarrow \left( \text{source}(e') == \text{cache} \land \text{type}(e') == \text{reply} \right)$$

If the user agent sends a request message for the first time, which means that its cache does not have the record of the resource, the user agent needs to get the resource from the server. Otherwise, if the response is cacheable, it will be stored in the user agent’s cache, and the user agent can get the resource from its cache directly without visiting the server. That is if two requests want to get the same cacheable resource, the second one can visit the cache directly instead of visiting the server.

Stateless

In REST architecture, the communication is stateless, i.e., each request message contains all the information needed to understand the message itself. Context storing in the server cannot be used and all the session states remain in the user agent. In our CSP model, the process SERVER is not modeled as the concurrent composition with the process Cache so that the session cannot be remained in the server. The format of the message sent by the user agent has been defined in the first part of Section 4.
and the message contains all the information needed to understand it in the four-tuple info_cons.

**Constraint 3:**

$$\forall tr \in \text{trace}(system) \bullet (system == \text{system}/tr \wedge (source(tr_0) == server \wedge type(tr_0) == reply))$$

Based on the traces analysis, the system will be back to the original state after it performs one session. Here, the process system will be back to itself after it has done the trace tr. The end mark of one session is that the server sends back a reply to the intermediary or the client.

**Uniform Interface**

In our CSP model, we map the Uniform Interface constraint to the communication channels. The messages transmitted on the channels are unified and the format of the messages has already been defined in the first part of Section 4. In general level, the message can be summarized as $$\{\text{msg}, \text{resource}, \text{format}, \text{sender}, \text{receiver}\}$$. Four constraints of the interface have been described in the four-tuple resource_format.

**Constraint 4:**

$$\forall tr \in \text{trace}(system), \forall e \in \text{set}(tr), \exists e' \in \text{set}(\text{trace(system}/tr)) \bullet (\text{type}(e) == \text{request} \wedge (\text{source}(e) == \text{client} \vee \text{source}(e) == \text{cache} \vee \text{source}(e) == \text{intermediary}) \implies \text{source}(e) == \text{target}(e') \wedge \text{source}(e') == \text{target}(e) \wedge \text{channel}(e) == \text{channel}(e') \wedge \text{type}(e') == \text{reply})$$

As mentioned above, for any event in any trace of the process system, if there is a request message sent through the channel by client, intermediary or the cache, after the process system has done the trace, there must exist an event e' which contains a reply message sent through the same channel. According to the mapping of the Uniform Interface constraint and the definition of the message, the sender of the request message must be the same as the receiver of the reply message.

**Layered and Code-On-Demand**

Layered constraint and Code-On-Demand constraint are obvious. We give the descriptions about how we map these two constraints to our model using Constraint 5 as follows:

**Constraint 5:**

$$\text{System} = \text{UserAgent} || \text{Intermediary} || \text{Server}$$

- **Layered.** In our model, we have the processes UserAgent, Intermediary, Cache and OriginServer. All the components are modeled as independent processes and can get the knowledge only from its neighbor components, which breaks the architecture into different levels. If a new service is added to the model, we can ab-
abstract the service as a new process and combine the new process with the process OriginServer without the effects of the user agent.

- **Code-On-Demand.** It is an optional constraint of REST architecture. We do not discuss more about it. Using concurrent operation, different processes are combined. Our model is easy to add new process in the system to expand the system.

5.2 REST Architecture in PAT

In this subsection, our model of the REST architecture is implemented using PAT. There are four main processes including UserAgent(info_cons), Cache(Node), Intermediary() and OriginServer(). We omit a large number of the PAT codes, only giving the typical relevant codes about the actions of the intermediary component as follows to show how we use PAT to implement our model:

```plaintext
Intermediary() = ComUC?msgreq.info_cons.UA.IC->
ComCIC!msgreq.info_cons.IC.CA->
ComCIC?msgrep.content.CA.IC->
Intermediary.Check(intermedia, content);

Intermediary.Check(info_cons, content) = ifa(content == 0)
{ComCS!msgreq.info_cons.IC.OS->
ComCS!msgrep.content1.OS.IC->
ComUC!msgrep.content1.IC.UA->
(cache_media_list[info_cons][1] = content1)-->
Intermediary()}
elsec
{ComUC!msgrep.content1.IC.UA->
(cache_media_list[info_cons][1] = content)-->
Intermediary()};
```

Due to we have already given formal definitions of the six REST constraints in Subsection 5.1 and we also give a detailed description about how we map these constraints to our model, here we do not list all the assertions of the REST constraints. We just give two examples about the assertions of Client-Server constraint and Cacheable constraint to show how to express these constraints using the syntax of PAT. Here is the assertions in Table 1. According to the formal descriptions as mentioned above, we verify the constraints using PAT and the results of the verification prove that our model caters to the REST architecture.

<table>
<thead>
<tr>
<th>Table 1. Assertions of REST Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>#assert System()=</td>
</tr>
</tbody>
</table>
| #define goal1 cache_user_list[0][1]=0;
| #define goal2 cache_user_list[1][1]=0;
| #define goal3 cache_user_list[2][1]=0;
| #define goal4 cache_user_list[3][1]=0;
| #assert System()=||(|ComUC!msgreq & (goal1||goal2||goal3||goal4))|->
| (||ComCS!msgrep & & ComUC!msgrep)); |
6 Case Study

In this section, a case study about a scenario for environment monitoring is given to illustrate the feasibility of our framework in modeling and analyzing the RESTful systems. It shows that our proposed framework is not only confined to HTTP but can also be applied to other REST-compliant protocols such as CoAP. More details about the scenario is shown in Figure 3.

The scenario is composed of a wireless sensor network (WSN), an environment supervision unit (ESU) node and an environment monitoring centre (EMC) node. The WSN is made up of numbers of wireless sensor nodes deployed with CoAP and used for the acquisition of values of some physical quantities, for example, the concentration of phosphorus in a river. The ESU is the main computing device which is the border router for the WSN and manages the communication with back-end EMC. EMC acts as a client to access data in those wireless sensor nodes for workers to make correct decisions to control the concentration of phosphorus for environmental protection. Additionally, a telematic device is used for information exchange with the Internet.

For simplicity, we only consider about the communications among a WSN node \( w \), an ESU node \( s \) and an EMC node \( m \) situated on the Internet. According to the model built in section 4, \( m \) stands for the user agent, \( s \) is an instance of the intermediary component and \( w \) denotes a server. In one interaction, the user agent \( m \) is intended to get a resource, i.e., the value of the concentration of phosphorus, stored in sensor node \( w \). Based on the traces analysis, we give the trace of the whole process as follows:

\[
\langle m.ComUIC!msg.request.m.s, s.ComUIC?msg.request.m.s, s.ComICS!msg.translated_request.s.w, w.ComICS.translated_request.s.w, w.ComICS!msg.response.w.s, s.ComICS?msg.response.w.s, s.ComUIC!msg.response.s.m, m.ComUIC?msg.response.s.m... \rangle
\]
At first, the EMC node \( m \) sends an HTTP GET (we use \( GET_{HTTP} \) for distinction of the GET method of CoAP, which is expressed as \( GET_{CoAP} \)) request to get the concentration of phosphorus of node \( w \). ESU node \( s \) or cache \( c \) can receive the request.

\[
\text{request} = (\text{URI} : \text{coap}://node-w.net/phosphorus, \text{repr} : \{\}, \\
\text{sdi} : \{\text{METHOD} : \text{GET}_{HTTP}, \text{other} - \text{functional} - \text{fields}\}, \\
\text{hmedia} : \{\text{hypermedia}\})
\]

If the resource is cacheable and the response stored in cache \( c \) is not out of time, then it can be sent to node \( m \) immediately and node \( m \) rejects duplicated responses arriving later. If it is neither a cacheable resource nor found in cache \( c \), \( s \) translates this HTTP request into a CoAP request and sends it to the WSN node \( w \).

\[
\text{translated} - \text{request} = (\text{URI} : \text{coap}://node-w.net/phosphorus, \text{repr} : \{\}, \\
\text{sdi} : \{\text{METHOD} : \text{GET}_{CoAP}, \text{other} - \text{functional} - \text{fields}\}, \\
\text{hmedia} : \{\text{translated} - \text{hypermedia}\})
\]

After processing the request, \( w \) returns a CoAP response and after numbers of hops, the response is finally received by node \( s \). This response may include some response codes indicating the result of the attempt to understand and satisfy the request. It can be described as follows.

\[
\text{response} = (\text{uri} : \{\}, \text{repr} : \{\text{a document of type application/link - format including a hyperlink coap}://node-w.net/calcium\}, \\
\text{sdi} : \{\text{cacheTag}, \text{functional} - \text{fields}, \text{hmedia} : \{\text{hypermedia}\})
\]

Node \( s \) receives this response, translates it into an HTTP response and sends it to node \( m \). This translation may involve protocol details which are not our concerns. So the translated response is not given. Using our framework, we want to provide a better understanding of REST architecture and its constraints. Here, the REST constraints are listed from the perspective of our model as follows:

- **The Client-Server Constraint.** Node \( m \) serving as a user agent and \( w \) as an origin server forms a client/server model.
- **The Stateless Constraint.** Since messages transmitted in the scenario are self-descriptive (see the message format of request) and there is no cache to store sessions in the server, this model is stateless.
- **The Cacheable Constraint.** We add a tag \( \text{cachTag} \) in the self-descriptive messages of a response message indicating whether a response is cacheable or not. We also enable node \( m \) and \( s \) to have the capacity of the storage of cacheable responses. This scenario satisfies the REST’s cacheable constraints.
- **The Layered Constraint.** With the border router \( s \), which is regarded as the intermediary component, added, this integrated network is intuitively layered.
- **The Uniform Interface Constraint.** We confine channels in the model to the transmission of several kinds of messages and we consider these channels as uniform interfaces. Since each message contains \( \text{URI} \) for identifications of resources, \( \text{representations} \) and the operation methods for operations of resources by manipulating representations, some meaningful header fields which can serve as the self-descriptive information and hyperlinks for the transition to next state, we can
conclude that channels, in this point, are mechanisms for the realizations of uniform interfaces. The four interface constraints are listed in Table 2 as follows:

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identification of Resource</td>
<td>URI: coap://node-w.net/phosphorus</td>
</tr>
<tr>
<td>Manipulations of Resources through Representations</td>
<td>Type: a document of type application/link-format METHOD: GET</td>
</tr>
<tr>
<td>Self-descriptive Messages</td>
<td>Protocol versions, cacheTag</td>
</tr>
<tr>
<td>Hypermedia as the Engine of Application State</td>
<td>hlink: coap://node-w.net/calcium</td>
</tr>
</tbody>
</table>

Using our framework, better understanding of the RESTful system in this case study scenario can be achieved and the REST constraints can be illustrated clearly, which demonstrates the feasibility of our model. Moreover, we also use model checker PAT to verify the model of this case study to show that it caters to the REST architecture.

7 Conclusion and Future Work

In this paper, we have proposed a CSP model of the REST architectural style. All of the components in the architecture, including user agent, cache, intermediary component (proxy and gateway) and server, have been specified as processes respectively. Besides, we have discussed the constraints of the REST architecture, including Client-Server, Cacheable, Layered, Uniform Interface, Stateless and Code-on-demand, and mapped these constraints to our achieved model. Moreover, we have used the model checker PAT to verify the constraints to prove that our achieved model is consistent with the REST architecture and detailed descriptions about how to describe these constraints to our framework also have been given. Finally, our model has been applied in modeling and analyzing a web application case study. Our formal framework makes a better understanding of REST from the perspective of formal methods to guide the implementations of the RESTful systems.

For the future, we will continue working on the modeling and analyzing of the REST architecture and the web service. Our model will be applied in reasoning about other web applications in the industry and real world such as the smart home and we also want to use our framework to determine whether a web application caters to REST architecture or not. Further, with formal tools, verifications based on our achieved model for REST architecture are also an interesting topic to be explored.

Acknowledgement The authors gratefully acknowledge support from the Danish National Research Foundation and the National Natural Science Foundation of China (Grant No. 61061130541) for the Danish-Chinese Center for Cyber Physical Systems. This work was also supported by National Basic Research Program of China (No. 2011CB302904), National High Technology Research and Development Program of China (No. 2011AA010101 and No. 2012AA011205), National Natural Science Foundation of China (No. 61021004), and Shanghai Leading Academic Discipline Project (No. B412).
References

SiteHopper: Abstracting Navigation State Machines for the Efficient Verification of Web Applications

Guillaume Demarty, Fabien Maronnaud, Gabriel Le Breton, and Sylvain Hallé

Department of Computer Science and Mathematics
Université du Québec à Chicoutimi, Canada

Abstract. A Navigation State Machine (NSM) is a conceptual map of all possible page sequences in a web application that can be used to statically verify navigation properties. The automated extraction of an NSM from a running application is currently an open problem, as the output of existing web crawlers is not appropriate for model checking. This paper presents SiteHopper, a crawler that computes on-the-fly an abstraction of the NSM based on link and page contents. Experiments show that verification is sped up by many orders of magnitude for applications of real-world scale.

1 Introduction

As web applications form an ever greater part of existing software, their weaknesses account for an increasing part of known vulnerabilities. The knowledge of an application’s navigational structure becomes crucial, especially as two of OWASP’s 2010 top ten web application security risks [12] are related to navigation handling. It hence becomes desirable to leverage existing formal verification techniques to web applications. In Section 2, we recall how the exhaustive exploration of all links contained in the pages produced by a web application induces a graph called a Navigation State Machine (NSM), which describes the set of all navigation traces expected by the application’s code.

We show how the possession of a machine-readable form of the NSM can be put to numerous value-adding uses, from runtime enforcement to static analysis. For example, it has recently been shown how traditional model checking techniques can be applied to this NSM to statically verify navigation properties expressed as temporal logic formulae [5, 7, 9, 14].

All these works take as given the NSM that models the application to be verified. The generation of the NSM is a step that has been consistently overlooked, on the grounds that so-called web crawlers (or spiders) can navigate the application and extract the list of links between all pages, from which an NSM

* We acknowledge the financial support of the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Fonds québécois de recherche sur la nature et les technologies (FQRNT).
can easily be built. However, we shall see in Section 4 that the output produced
by classical crawlers is far from appropriate for verification purposes, thereby
revealing an important missing link in the process.

We present in Section 5 a web crawler called SiteHopper, aimed at producing a
formal model of an application’s navigational structure suitable for consumption
by automated verification tools. In particular, SiteHopper computes on-the-fly,
as the application is being explored, an abstraction of its pages and links aimed
at reducing state redundancy and model size. Section 5.2 reports our findings
on the use of SiteHopper on real-world applications; it presents the potential to
greatly simplify the resulting NSM and extend the reach of model checking tools.

Finally, in Section 6, we explain why existing tools such as web crawlers
lack some of the features required for the extraction of a precise and meaningful
site map. Therefore, to the best of our knowledge, SiteHopper’s a symbolic
management of URL parameters is original.

2 Navigation Sequences in Dynamic Web Sites

In this section, we shall illustrate how many bugs and vulnerabilities in web
applications turn out to be related to an inadequate handling of constraints on
navigation paths. To this end, we designed a web site that allows users to browse
a list of items from a fictitious company’s catalogue called “Hardware Joe”.

The site’s main page (index.php), shown in Figure 1, provides a simple
description of the company and a link to a contact page. The user must fill a
form, providing a login and a password to access the site’s catalogue. Provided that
proper credentials are given (user-home.php) the user can then browse the main
list of items (item-list.php). The list provides links to a page (item-info.php)
that gives detailed information on each item, given a specific item id. On this
page, the user can return back to the list, purchase the item or edit the item’s
information. Each of these pages also have a clickable “Logout” link that takes
the user to a logout confirmation, and then back to the home page.

2.1 Hypotheses on Web Applications

Albeit simple, the site embodies a number of implicit hypotheses about the
structure of a web application, which will be used in the method presented in the
next section. First, the application exposes its functionality through pages, which
are distinct unit of processing that provide a single, well-defined functionality. This
unit is accessed by sending an HTTP request to a unique base URL. Moreover,
the invocation of a URL ends either in the production of HTML content to be
displayed by the user’s browser, or in the redirection to another resource.

Upon being called, a resource can use additional context information to
perform its task. In the present case, it is assumed that this context comes from
two sources: parameters passed along with the HTTP request that invokes the
URL, and the current page from which the URL is being called. In the Hardware
Joe example, the item-info page uses an id parameter to determine what item
Fig. 1. Start page of “Hardware Joe”, the example web site

to show information about. It shall be noted, however, that the context modulates, but does not fundamentally changes the page’s core functionality.

As a rule, valid navigation paths form a subset of all possible sequences of page calls, and users must therefore follow a set of constraints during their navigation. For example, the front page of Hardware Joe does not show a link to item-buy, since a user must first be logged and choose an item id for this action to be meaningful and valid. Similarly, no Logout link is displayed in pages unless the user has previously logged in.

The annotated graph encoding all valid navigation sequences is called a Navigation State Machine (NSM), where each page is a vertex and an edge A→B indicates that page A contains a link to page B. Formally,

Definition 1 A Navigation State Machine is a tuple $M = (N, P, V, S, s_0, \delta)$, where $N$ is a set of page names, $P$ is a set of parameter names, $V$ is a set of parameter values. $S \subseteq N \times (P \rightarrow V)$ is the set of states, where each state is defined by a page name and a mapping from parameters to values. As usual, $s_0 \in S$ is the initial state, and $\delta \subseteq S^2$ is a transition relation between states.

We will use the notation $\delta(s)$ to denote the set $S' \subseteq S$ such that $s' \in S'$ if and only if $(s, s') \in \delta$.

2.2 Purpose of NSMs

The NSM shows not only how each page is linked to others, but more importantly what the application expects the flow of navigation to be. However, browser functionalities such as the “back” button and the use of multiple windows and bookmarks, allow users to request pages from web applications in unpredictable ways. It was shown in an earlier work [9] that the flaws exposed above can bear consequences that range from the simple inconvenience to a serious security breach. It was argued, however that they share a common cause: the user does
not follow the intended flow of navigation, which in turn causes assumptions about the application’s state to fail and produce undesirable behaviour. Hence the possession of the NSM for a given web application opens the way to a variety of uses.

A first possible use is to make sure that the developed application follows a specification that was given beforehand. Moreover, if the NSM is a faithful abstraction of the application’s navigation paths, one can perform static analysis on the NSM itself rather than on the actual application’s code. One can use model checking tools to verify properties of navigation paths expressed in temporal logic [5, 7, 9, 14]. It hence becomes possible to formally demonstrate, for example, that on every valid navigation trace, a user that logs in will always eventually reach the logout page.

A second possible application is to use the NSM in a prescriptive manner, and to enforce at runtime that a user sticks to the paths that the map stipulates. Previous work by Hallé et al. developed a plugin for PHP applications based on the Zend or CodeIgniter frameworks [9]. This plugin acts as an additional layer on top of an existing application that keeps track, upon each page call, of the current state of the NSM. In the event a user veers outside the paths that are stipulated, the plugin prevents the page from being loaded and simply re-displays the last page.

Some of OWASP’s Top-10 web application vulnerabilities for 2010 [12] can be prevented in this manner. For example, vulnerability A4, “Insecure Direct Object Access”, can happen when a user modifies a parameter in a URL to a value not authorized in the current state of the application. Assuming the application only provides links with authorized parameter values, it is impossible to summon such a page by staying inside the NSM.

We can finally use the NSM to automatically drive test sequences on the actual application. With the knowledge of links between pages, it becomes possible to generate valid navigation sequences that take the user to a given page, for debugging purposes, as was attempted in [16], or tools like ReWeb [13] and TestWeb [15].

3 A Review of Crawler-Generated NSMs

All the aforementioned applications take the state machine model as given. It is therefore sensible to first harvest a navigation map out of existing crawlers such as Googlebot, Microsoft’s Bingbot, Heritrix, Nutch or VeriWeb [4].

3.1 Model Checking Crawler-Generated NSMs

Classical crawlers are designed to systematically follow page links in a web site with the purpose of building some form of index. They use a page’s URL as the unique identifier for each node. In the case of the Hardware Joe application, a “classical” analysis of page links leads to the NSM shown in Figure 2, where page names and parameter-value pairs are shown in URL style.
Fig. 2. The navigation state machine for the example web site. Links to `logout` from the various items pages are omitted for clarity.

In our example, since the item list is generated dynamically from the contents of a database, there are as many “Item Info”, “Item Buy” and “Item Edit” pages as there are items in the database —and consequently, as many nodes in the resulting NSM that these crawlers produce.

A simple script we designed can convert a Google SiteMap\(^1\) file into a NuSMV state machine. Table 1 shows the time required by NuSMV to validate the CTL property “the user can always return to page \(p\)” (AG EF \(p\)), on our example’s NSM with an increasing number of items in the application’s database. While the number of nodes in the NSM and the size of NuSMV’s input specification grows linearly with the number of items, validation time increases much faster; NuSMV fails to process the NSM when the application’s catalogue hosts 10,000 items.

3.2 Issues with Hard-Coded Parameters

The tight coupling of catalogue items and NSM nodes is problematic when one is to use a crawler’s output to perform static verification of navigation properties. Apart from rendering the problem intractable for realistic page counts, this characteristic is the source of other problems.

First, if the `item-list.php` page is generated dynamically from the contents of a database, the list of available items can become much larger than the five

\(^1\) [http://www.sitemaps.org/](http://www.sitemaps.org/). The format describes a standardized way of listing a web application’s pages.
Table 1. Validation time for increasing database size in a web store application. NuSMV crashes with a segmentation fault at 10,000 items.

<table>
<thead>
<tr>
<th>Nb. of items</th>
<th>Input size (kB)</th>
<th>Time (s)</th>
<th>Nb. of items</th>
<th>Input size (kB)</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1.940</td>
<td>0.009</td>
<td>300</td>
<td>176.4</td>
<td>0.365</td>
</tr>
<tr>
<td>10</td>
<td>6.534</td>
<td>0.018</td>
<td>1,000</td>
<td>587.9</td>
<td>2.688</td>
</tr>
<tr>
<td>30</td>
<td>18.16</td>
<td>0.034</td>
<td>3,000</td>
<td>1,777</td>
<td>22.552</td>
</tr>
<tr>
<td>100</td>
<td>58.83</td>
<td>0.084</td>
<td>10,000</td>
<td>5,940</td>
<td>—</td>
</tr>
</tbody>
</table>

id's shown in the figure. This will result in a large number of outgoing links, each of which will need to be explored separately to obtain the complete site map, thereby significantly increasing the time required to exhaust the whole site's contents. In addition, the size of NSMs has a direct impact on the performance of many of the uses of an NSM described in Section 2.2.

Second, providing hard values of the id parameters in the state machine amounts to hard-coding the database contents in the site map. This coupling between navigation and data content makes the resulting NSM highly vulnerable to any changes in the store's catalogue. For example, adding a new item in the database will result in one new set of links and pages being available to the user, yet the NSM won't reflect that change unless the application is crawled again. The runtime enforcement or static verification of an application using an outdated NSM may result in false conclusions that undermine their usefulness.

Third, each independent exploration of the site with a different value of id will result in the duplication of the same link structure between a small number of pages. Such repetition of what is essentially the same set of pages provides little insight about the site's actual structure, and in particular does not highlight the fact that the trio of pages item-info, item-buy and item-edit always behave in the same way and merely take an id as an argument.

It shall be noted upfront that page parameters, such as item IDs or user names, cannot be discarded altogether, as some web crawlers do. Indeed, in our example application, there exists a link between the “View” and the “Edit” page for the same item ID, but no link if the ID is different. Page parameters must be taken into account to retain a faithful navigation map suitable for verification.

4 A Symbolic Management of Request Parameters

These cheerless results are surprising in that the largest NSM in Table 1 is small according to model checking standards (30,000 states). The culprit rather comes from the fact that a very large part of an NSM's size is made of the repetition of patterns of essentially similar pages expressed in extenso. Moreover, the information provided by the crawler is not sufficient to assess which pages are similar and could be abstracted; the page's contents also matters. For all these reasons, the simplification of the NSM must be performed upstream of the model checker, at the link harvesting step.
In this section, we expose a technique to automatically discover and handle parameters in application links symbolically. The technique operates on-the-fly as the pages are explored, and can hence be integrated into a web crawler, as we shall see in the next section.

4.1 Abstracting multiple parameter occurrences

Our abstract crawler attempts to generalize the structure of pages in an NSM by replacing multiple nodes (i.e. pages) containing hard-coded parameter values with a single node. At the heart of the abstraction mechanism is the concept of symbolic parameter, which acts as a placeholder for actual values, and upon which constraints can be expressed.

The first component of the method consists in detecting sets of links inside a page that appear to be programmatically generated, and hence present high odds of being populated with answers from a database query. A set of such links is likely created by some program loop, and hence the URL structure of each of these links will be similar, except for the values of the “machine-generated” parameters.

When such a pattern is detected in a page’s code, we abstract the links that fit the pattern by replacing them with a single node in the NSM, where the repeated parameter is replaced by a symbolic variable and associated with the set of values recorded for that parameter in the explored page.

We extend Definition 1 by adding to $V$ a disjoint set $V_s$ representing symbolic values. Formally, for some parameter $p \in P$ and two nodes $s = (n, f)$ and $s' = (n, f')$ such that $f(x) = f'(x)$ for all parameters $x \in P \setminus \{p\}$, we replace $s$ and $s'$ by $s'' = (n, f''')$, where $f''(x) = f''(x) = f'''(x)$ for all parameters $x \in P \setminus \{p\}$, and where $f'''(p) = v_s$ for some fresh symbolic value in $V_s$. The new transition relation $\delta''$ is built such that $\delta''(s'') = \delta(s) \cup \delta(s')$, and for all $y \in S$, $\delta''(y) = s''$ whenever either $\delta(y) = s$ or $\delta(y) = s'$.

When visited by the crawler, a page with symbolic parameters needs to be instantiated —that is, hard values must obviously be used in place of symbolic parameters.

The abstraction of all links with different parameters values into a single one with a symbolic variable assumes that all pages resulting from giving a value to this parameter perform essentially the same functionality. We recall that, consistent with the hypotheses laid out in Section 2.1, a parameter passed to a page modulates its functionality, but not in an essential way. This entails, in particular, that exploring the page with some parameter value $k$ will produce links that are similar (up to their own parameter values) to those one would find by exploring the same page with another value $k'$.

The abstraction scheme also presents the advantage of providing a graceful way of dealing with input forms. All their fields are actually parameters to the POST request that is sent when a user clicks on their Submit button. Many users can type in many values to these fields, and each such form will result in a request URL that only differs from others in its user-supplied values. Therefore,
a form is no different than a list of program-generated links, and its parameters can be marked as free in the same way.

4.2 Context-Based Abstraction

The abstraction of all links with different parameter values into a single one with a symbolic variable assumes that all pages resulting from giving a value to this parameter perform essentially the same functionality. However, for many real-world web applications, the URL alone does not provide enough information to properly discriminate similar links; for example, many web application frameworks use the same page and parameter names for all of their links.

To address this issue, we refine the previous abstraction mechanism by also analyzing the context where the links appear. Links leading to similar pages are generally grouped in the same logical section within a page, such as the same top-level container element. For two links that are candidates for abstraction according to the previous rule, the path to their closest common ancestor in the page’s DOM tree is computed. Depending on the contents and the length of that path, the decision for performing node abstraction can be vetoed. Moreover, this veto function is user-definable, and can be fine-tuned depending on the application to analyze.

Formally, this is represented by defining a function \( d : S \rightarrow T \), where \( T \) is the set of HTML DOM trees. The function \( d \) is built on-the-fly, and each page is associated to its DOM tree once visited by the crawler. The veto described above is a function \( v : T^2 \rightarrow \{ \top, \bot \} \) that takes two DOM trees \( t \) and \( t' \) and evaluates their DOM similarity. The fusion of the nodes, as per the previous step, is only performed if \( v(d(s), d(s')) = \top \).

4.3 Matching input and output parameters

Often times, sequences of pages carry the same parameter in their URLs. This is the case in our example application, where loading the info page for some item ID \( \text{off} \) ers links to the “Buy” and “Edit” page, with the same ID. Hence the inclusion of symbolic parameters reduces the number of nodes in the NSM, but is not sufficient to retain the correct structure of the web application, as a page with a symbolic ID parameter does not say anything about the relationship this ID has with the next page.

A further simplification step addresses that issue. When the crawler loads a page \( s_1 = (n_1, f_1) \) where some parameter \( p \) is instantiated with value \( k \), and detects that the links to some other page \( s_2(n_2, f_2) \) are such that \( f_2(p) = k \), then in \( s_2 \) we set \( f_2(p) \) to the same (symbolic) value as defined by \( f_1(p) \). This indicates that, when loading page \( s_2 \), the value to give for parameter \( p \) should not be arbitrary, but rather propagate the value that was used to load the previous page, \( s_1 \). We currently detect parameters with equal values, but the process could be enhanced to discover simple relations such as integer increments, by defining the transition relation between as a set of Boolean conditions expressed on source
and target $n$ and $f$ (the previous example would have $(n = n_1 \land n' = n_2) \rightarrow f'(p) = f(p)$).

As a side benefit, a web crawler using such a symbolic management of dependencies between URL parameters is also immune to some forms “spider traps”, i.e. potentially infinite sequences of pages being dynamically generated by a script. A simple example of a spider trap is a web calendar providing on each page a link to the previous and next month. A crawler that does not abstract request parameters will not discover the redundancy and treat each page as a distinct one, infinitely looping backward and forward until resources are exhausted. In contrast, the symbolic management of parameter dependencies across pages will result in a single page that only requires to be explored once.

4.4 Domain Reduction

The previous simplification then allows us to perform significant reductions in the size of domains for observed parameters. For example, in our online store application, there is no other constraint on item IDs than the fact that the ID passed to the “Item Buy” and “Item Edit” page must be the same as the ID from the source “Item Info” page. The actual value of the ID is never relevant in the navigation, which entails that the domain for the symbolic parameter ID can be shrunk down to a single constant. For verification purposes, only “hard” values of parameters occurring in pages actually matter, even if the domain for this parameter is larger in practice. We take advantage of this by trimming from the set $V$ all values that do not show up in any node’s image for their mapping $f$.

5 Empirical Evaluation

To test the parameter abstraction scheme described above, we created a web crawler called Site Hopper, which can be used to automatically build and simplify an NSM, and used it in a number of experiments which we describe in this section.

5.1 Crawler Implementation

SiteHopper computes on-the-fly, as the application is being explored, an abstraction of its pages and links aimed at reducing state redundancy and model size by deriving meaningful relationships on link parameters inside and between pages. It operates like a traditional crawler, at the HTTP protocol level, by only sending requests and analyzing the contents of returned pages. In particular, it does not require the application’s source code; therefore, SiteHopper is not tied to any particular web programming language, and can be used to explore any web application that a normal browser could access. SiteHopper 1.0 is freely available under an open source license and has been downloaded more than 90 times.

The crawler presents itself to the user in the form of an interactive web application, whose interface is shown in Figure 3. A user first types in the page’s
top textbox the starting URL. Site Hopper then loads that page, analyzes its contents and displays in the lower region of the page a partial graph showing the pages and links between them that have been discovered so far. The graph is interactive: by clicking on one of the nodes, the user resumes the exploration from that particular page, and any new pages and links discovered are added to the graph on-the-fly.

SiteHopper is made of two components. The first part, invisible to the user, is called the \textit{jumper}: it consists of a web service, hosted on Site Hopper’s server, responsible for issuing a particular HTTP request, retrieving the page contents and analyzing the links it contains. The jumper is built to return responses in an XML format extending Google’s SiteMap protocol. The second part of Site Hopper is the GUI itself, an Ajax application running in the user’s web browser and displaying the progressive construction of an application’s NSM. To create a state machine out of the jumper’s data, the GUI persists in memory the graph structure created by merging responses to successive requests to the jumper. The resulting structure is again an extended SiteMap XML document that the user can save to a file at any moment.

5.2 Experiments

To assess the effectiveness of our method at generalizing navigation state machines from web applications, we ran Site Hopper on a set of real-world applications.
Besides Hardware Joe, the web site used as an example throughout this paper, we ran Site Hopper on a number of applications:

- Digitalus, an open source content management system
- osCommerce, an open source online catalog management system (8,000 lines of code)
- Phreebooks, an open source inventory management system.

Phreebooks is notable for being hosted online, on a server to which we had no special access outside of the web interface available to anybody to try the application.

**NSM extraction** We first report the capability of the tool to explore the applications, correctly request their pages and provide meaningful information on symbolic parameters and links between pages.

**Phreebooks: Bootstrap Script** In the case of Phreebooks, all the links in the application point to a single base URL, `index.php`, that acts as a “bootstrap script”, which is then responsible of dispatching it to the proper piece of code. No matter what action or what resource is being handled: everything is passed as parameters to this single index file. Hence, editing user account `guillaume` would have a URL `index.php?module=users&page=edit&id=guillaume`, while showing the list of all blog posts on the topic “computers” would have the URL `index.php?module=blog&page=show&cat=computers`. Site Hopper generalizes the site as having a single page of the form `page.php?module=$i&page=$j&cat=$k&id=$l...`, with $i$, $j$, $k$ and $l$ free input parameters.

The basic processing unit is therefore not identified by the base URL, but the value of the `module` and `page` parameters. This goes against our assumptions that parameters do not modify the functionality of a page in a fundamental way. Site Hopper can be adapted so that it can take the page name and some parameters as the identifier of what constitutes a distinct “page”. Once told that `module` and `page` should be handled as part of the page name instead of parameters, Site Hopper had no trouble exploring the application and abstracting the remaining parameters correctly.

**Digitalus: Clean URLs** Digitalus exposes “clean” URLs to the user using Apache’s `mod_rewrite` module, which allows applications to rewrite URLs on-the-fly upon being invoked. For example, the application can specify a rewriting rule that transforms the URL `edit/users/guillaume` into `index.php?page=edit&id=guillaume` on the server side, and the corresponding script be called.

A configuration file allows Site Hopper to handle such a URL naming scheme. The user provides a predefined list of character strings that define page names,

---

2 http://www.digitaluscms.org
3 http://www.oscommerce.com
4 http://www.phreesoft.com
such as edit/users. When encountering a link, Site Hopper looks in the list for the occurrence of such a string, which makes up the “page” part of the URL. The remaining, slash-separated elements are considered as nameless parameter values. This simple mechanism again turned out to work surprisingly well: Site Hopper correctly interacts with the application and manages parameters symbolically.

These experiments show that the adaptations required to our crawler are not related to our parameter abstraction method, but rather to the capability of distinguishing what constitutes a page vs. its parameters. Once this distinction is made clear, the parameter abstraction mechanism works correctly in all applications we tested.

**Empirical Data** The combined application of the abstraction steps presented in Section 4 is expected to yield significant reductions in the size of the NSM, while retaining the same navigation sequences. We illustrate the effectiveness of our method at generalizing navigation state machines by running SiteHopper on our example application; the results are shown in Table 2. One can see in the first time column that validation is on average 20 times faster than with an NSM generated by a traditional crawler, due to the fusion of all nodes with different item IDs into a single node. Validation is further improved when adding domain reduction, which, in this case, reduces the NSM into a graph of constant size, regardless of database size. For a database of 3,000 items, this represents a 2,000× speedup with respect to the initial NSM.

<table>
<thead>
<tr>
<th>Nb. of items</th>
<th>Time 1 (s)</th>
<th>Time 2 (s)</th>
<th>Nb. of items</th>
<th>Time 1 (s)</th>
<th>Time 2 (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0.009</td>
<td>0.011</td>
<td>300</td>
<td>0.018</td>
<td>0.011</td>
</tr>
<tr>
<td>10</td>
<td>0.010</td>
<td>0.011</td>
<td>1,000</td>
<td>0.158</td>
<td>0.011</td>
</tr>
<tr>
<td>30</td>
<td>0.007</td>
<td>0.011</td>
<td>3,000</td>
<td>1.427</td>
<td>0.011</td>
</tr>
<tr>
<td>100</td>
<td>0.008</td>
<td>0.011</td>
<td>10,000</td>
<td>16.14</td>
<td>0.011</td>
</tr>
</tbody>
</table>

Table 2. Validation time for increasing database size in a web store application with SiteHopper, using node fusion (column 1), and adding domain abstraction (column 2).

We repeated the analysis with osCommerce 2.3.1; we populated the application’s database with 10,000 elements, and then performed empirical measurements on two aspects of SiteHopper, its running time and the impact of the use of symbolic parameters on an NSM’s complexity.

**Running Time** Running time was defined as the time for the server-side part of SiteHopper to request and process a single page. The analysis step typically takes on average 700 ms per page, with pages containing more links taking a longer time to analyze. We shall mention that this figure includes the time where the web application itself processes the request and generates the page that is then analyzed by SiteHopper. As a rule, each link in a page adds on average 33 ms to the processing time. The total time to generate the whole NSM we used was on average 12.49 s.
Symbolic Parameters To assess the simplification potential of symbolic parameters and the mechanism described in Section 5, we performed a second experiment where we verified CTL properties on the symbolic NSM. The results are shown in Table 3. Validation time averages less than 0.5 s and is independent of database size, while the same validation with the explicit-state NSM crashes NuSMV for all properties, due to its size.

<table>
<thead>
<tr>
<th>Property</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main page is reachable from all pages</td>
<td>0.480</td>
</tr>
<tr>
<td>Product ID can only change through visiting the item list</td>
<td>0.421</td>
</tr>
<tr>
<td>Category ID is never constrained</td>
<td>0.415</td>
</tr>
</tbody>
</table>

Table 3. Validation time for a sample set of CTL properties

6 Related Work

Many specialists maintain the idea that crawling the web has been solved a decade ago. This contrasts with a 2009 exhaustive survey of program comprehension through dynamic analysis, which notes that web applications are the least studied of all types of applications on this respect [6]. Indeed, despite the broad range of tools developed over the past decade, we shall see that none of them can produce a site map and deal with symbolic parameters on-the-fly.

6.1 Source Code Analysis

A first approach consists of obtaining the application’s source code, and deduce valid navigation paths from the code’s control flow. This is the approach taken by Guha et al. [8], who statically analyze an Ajax application on the client side and extract a control-flow graph from its source code. Similarly, Licata and Krishnamurthi present a method for extracting a form of context-free grammar, called WebCFG, from a web application’s source code [10]. However, a WebCFG does not have support for symbolic parameters and cannot express dependencies between parameters across multiple pages. This is also true of a reverse-engineering approach called WARE [11].

6.2 Trace Mining

PHP2XMI [1], WAFA [2] and WANDA [3] work on the server side and can be classified as trace mining tools: they record information about page calls made by visitors by instrumenting the application’s source code, and process this log a posteriori. Some of them try to slice that information into traces corresponding to successive calls made by a single user. However, these tools produce sequence diagrams for individual visits, and do not actively explore an application to
obtain a map of all possible navigation sequences. PHP2XMI does apply some form of “filtering” to simplify the resulting navigation information; however, this filtering still considers as different the same base URL with different request parameters.

6.3 Site Crawlers

The first two approaches do not attempt to extract a model through the exhaustive exploration of the application; both rely on its source code, are tied to one specific implementation language, and assume knowledge of its internals, such as session variables. A natural choice for the extraction of navigation models is therefore the use of spiders or crawlers. Generally, web crawlers such as Googlebot or Microsoft’s Bingbot create unordered lists of pages, without keeping information on the way they are connected. While some of them can automatically populate form fields and access pages that require input parameters, they do not use that information to derive relationships between them in the way symbolic parameters in an NSM do.

An exception is a tool called VeriWeb [4], which, while not being labelled a “crawler”, explores interactive web sites using a special browser that systematically explores all paths up to a specified depth. However, VeriWeb, like classical crawlers, does not keep track of relationships and constraints in input parameters between multiple pages.

7 Conclusion

In this paper, we developed and exposed a technique to automatically discover and handle parameters in application links symbolically. The technique operates on-the-fly as the pages are explored, and can hence be integrated into a web crawler. The Site Hopper tool we presented in this paper was used to experiment on this method. Tests on real-world applications show that symbolic management of parameters results in a much clearer map of an application where abstract relationships between pages and parameters are explicitly shown. The method drastically reduces the size of the resulting navigation state machine and greatly decreases the adverse impact of database size on the performance of model checking of an NSM. It applies equally well on various URL schemes used by applications, provided that parameters can be distinguished from an URL’s base name.

Drawing on this success with server-side applications, the technique is currently being ported inside a web browser plugin, which opens the door to the analysis and abstraction of links generated by JavaScript on the client side. Future work also involves using measurements on the content of destination pages to decide whether to fusion outgoing links, and the study of bisimulation-based methods to identify redundant link structures in an NSM.
References


Preference and Similarity-based Behavioral Discovery of Services*

Farhad Arbab\(^1\) and Francesco Santini\(^{1,2}\)

\(^1\) Centrum Wiskunde & Informatica, Amsterdam, Netherlands
\[\text{Farhad.Arbab,F.Santini}@cwi.nl\]
\(^2\) Dipartimento di Matematica e Informatica, Università di Perugia, Italy
\[\text{francesco.santini@dmi.unipg.it}\]

Abstract. We extend Constraint Automata by replacing boolean constraints with semiring-based soft constraints. The obtained general formal tool can be used to represent preference-based and similarity-based queries, which allow a user more freedom in choosing the behavior of the service to finally use, among all possible choices. A user states his preferences through a “soft” query, and obtains results that satisfy this query with different levels of preference. The soft requirements may involve a parameter data of the service operations, or the (names of the) operations themselves. Moreover, we introduce a first implementation of the search procedure by using declarative (soft) Constraint Programming.

1 Introduction

Service-orientation is a design paradigm to build computer software in the form of services. The term “service” refers to a set of related software functionalities that can be reused for different purposes. In this sense, the service becomes more important than the software. A Service-Oriented Architecture (SOA) offers some benefits as return on investment, organisational agility and interoperability as well as a better alignment between business and IT. In such loosely-coupled environments, the automatic discovery process becomes more complex, and a user’s decision has to be supported taking into account his (often not crisp) preferences, some semantic information on the related knowledge-domain, and the behavior signature of each service, describing the sequence of its operations [17,11]. For instance, a user may need to find an on-line purchase service satisfying the following requirements: \(i\) charging activity is before shipping activity, \(ii\) to purchase a product, the requester first needs to log into the system and finally log out of the system, and \(iii\) filling the electronic basket of the user may consist of a succession of “add-to-basket” actions (a similar scenario is proposed in [17]).

In this paper, we define a formal framework that considers both user’s preferences and (stateful) service behavior during the search procedure, in order to

---

* This work was carried out during the tenure of the ERCIM “Alain Bensoussan” Fellowship Programme. This Programme is supported by the Marie Curie Co-funding of Regional, National and International Programmes (COFUND) of the European Commission.
retrieve multiple results for the same preference-based query; in this way, the end user has the possibility to choose among different results by selecting the service that maximizes his requirements. In the above mentioned purchase scenario, for example, he may prefer to pay with a credit card instead of with a bank transfer. Later in this paper, using the same framework, we also show how it is possible to represent similarity-based search, in order to find the services that perform operations “similar” to those requested. These services can be valid alternatives for the user in case the “best” (i.e., user’s most desirable) service is not available at the moment, for example, due to failures or a high number of requests.

In Sec. 2 we report the related work, showing that no general formal framework has been proposed in the literature for such tasks, and this dearth is even more striking if we consider the behavior of the services.

In Sec. 3 we summarize the background on semiring-based soft constraints \[6,5\], showing the basic operations of this parametric preference-based framework.

As a first result of this paper, in Sec. 4 we extend Constraint Automata (CA) \[2\] in order to deal with preferences on data-constraints: instead of (classical) boolean constraints, we adopt semiring-based soft constraints, which can model any preference system as long as it can be cast into a semiring algebraic structure. However, even boolean constraints can be represented with semirings (see Sec. 3), and used in the same framework as well.

In Sec. 5 we show how to model preference-based queries according to the theory presented in Sec. 4. The names of the service operations offered to users correspond to the names on the CA transitions (i.e., the synchronization constraints \[2\]). At the same time, soft data-constraints model preferences on the data exchanged through I/O by each service operation. For example, the data required for the Charging operation can involve a bank transfer, a credit card number, or pay-on-delivery with cash. For instance, a user may prefer the second method over the other two. Automata have emerged as a convenient framework to study behavioral issues. In Sec. 5 we also show that the formal results of Sec. 4, such as the join/hide operations on automata, and simulation/bisimulation relationships, can be used to reason on preference-based queries.

In Sec. 6 we focus our attention on similarity-based search. In this scenario, a user is interested in retrieving services with operations “similar” to those in his query; therefore, he uses soft constraints to define a preference for the operation names, and instead of their I/O data as in Sec. 5. For example, instead of the Charging operation, an operation named SendEstimate can be used by the same user to receive a purchase-estimate (and then buy with a phone call). In this case, the operations Charging and SendEstimate are “similar” and, then, mutually replaceable up to that user’s degree of preference. In Sec. 6.1 we show how to solve this search problem as a Soft Constraint Satisfaction Problem (SCSP) \[6,5\].

We suppose the availability of meaning and similarity-scores of names as computed from a proper domain-specific ontology \[15\] (as proposed by other works in Sec. 2): in this paper we focus on the representation of preference/similarity-based queries, and on the formal framework we propose to resolve them. Finally, in Sec. 7 we draw our conclusions and explain our future work.
2 Related Work

A handful of researchers have investigated the problem of business process reuse based on process similarity, and discovery of processes based on search through repositories. For example, in [3] the discovery queries are abstracted and dependencies among processes are described with the help of ontologies.

In [17] the authors propose a new behavior model for Web Services (WSs) using automata and logic formalisms. Roughly, the model associates messages with activities and adopts the IOPR model (i.e., Input, Output, Precondition, Result) in OWL-S [15] to describe activities. The authors use an automaton structure to model service behavior. They develop a new query language to express temporal and semantic properties of service behaviors. As a query evaluation algorithm they show an optimization approach using tree structures and heuristics to improve performance. However, similarity-based search is not mentioned in [17].

The model presented in [18] relies on a simple and extensible keyword-based query language and enables efficient retrieval of approximate results, including approximate service compositions. Since representing all possible compositions and all approximate concept references can result in an exponentially-sized index, the authors investigate clustering methods to provide a scalable mechanism for service indexing. In [10] the problem of behavioral matching is translated to a graph matching problem, and existing algorithms are adapted for this purpose.

In [4], the authors propose a crisp translation from interface description of WSs to classical crisp Constraint Satisfaction Problems (CSPs). This work does not consider service behavior and, in this framework, it is not possible to quantitatively reason on similarity/preference involving different services: it is not possible to widen the results of a query by obtaining similar services. In [20], a semiring-based framework is used to model and compose QoS features of WSs. However, no notion of similarity relationship is given in in [20].

In [8], the authors propose a novel clustering algorithm that groups names of parameters of web-service operations into semantically meaningful concepts. These concepts are then leveraged to determine similarity of inputs (or outputs) of web-service operations. In [16] the authors propose a framework of fuzzy query languages for fuzzy ontologies, and present query answering algorithms for these query languages over fuzzy DL-Lite ontologies.

In [11] the authors propose a metric to measure the similarity of semantic services annotated with an OWL ontology. They calculate similarity by defining the intrinsic information value of a service description based on the “inferenceability” of each of OWL Lite constructs. The authors of [12] present an approach to hybrid semantic matching of web-services that complements logic-based reasoning with approximate matching based on syntactic Information-Retrieval-based computations. In [19], the authors propose a retrieval method to assess the similarity of available service interfaces with a provided desired-service-description, extended to include semantically similar words according to wordNet.

Our solution in this paper appears to be more general and comprehensive compared to the works mentioned above, since it can be adapted to any semiring-like metrics, and comes with several formal tools for reasoning over the queries.
(see Sec. 5). Moreover, most of the proposed works do not consider the service behavior at all, but only their interfaces, because no formal standard for web-services covers their behavior yet. We propose an implementation based on (Soft) Constraint Programming, which proves to be expressive and efficient, and adopts (off-the-shelf) AI-based solving techniques in its underlying machinery.

3 Semirings and Soft Constraint Satisfaction Problems

A c-semiring [6] (simply semiring in the sequel) is a tuple $S = (A, +, \times, 0, 1)$, where $A$ is a possibly infinite set with two special elements $0, 1 \in A$ (respectively the bottom and top elements of $A$) and with two operations $+$ and $\times$ that satisfy certain properties over $A$: $+$ is commutative, associative, idempotent, closed, with $0$ as its unit element and $1$ as its absorbing element; $\times$ is closed, associative, commutative, distributes over $+$, $1$ is its unit element, and $0$ is its absorbing element. The $+$ operation defines a partial order $\leq_S$ over $A$ such that $a \leq_S b$ iff $a + b = b$; we say that $a \leq_S b$ if $b$ represents a value better than $a$. Moreover, $+$ and $\times$ are monotone on $\leq_S$, $0$ is the min of the partial order and $1$ its max, $(A, \leq_S)$ is a complete lattice and $+$ is its least upper bound operator (i.e., $a + b = \text{lab}(a, b)$) [6].

Some practical instantiations of the generic semiring structure are the boolean $\langle \{\text{false, true}\}, \lor, \land, \text{false, true}\rangle$, fuzzy $\langle [0..1], \max, \min, 0, 1\rangle$, probabilistic $\langle [0..1], \max, \times, 0, 1\rangle$ and weighted $\langle \mathbb{R}^+ \cup \{+\infty\}, \min, +, \infty, 0\rangle$ (where $\times$ and $+$ respectively represent the arithmetic multiplication and addition).

Given a semiring $(A, +, \times, 0, 1)$ and $a, b \in A$, we define the residuated negation [7] of $a$ as $\neg a = \max\{b : b \times a = 0\}$, where max is according to the ordering defined by $+$ [7]. Note that over the boolean semiring the negation operator corresponds to the logic negation, since $\neg 0 = \max\{b : b \times 0 = 0\} = 1$, and $\neg 1 = \max\{b : b \times 1 = 0\} = 0$. When the considered semiring has no $0$ divisors (i.e., when $a \times b = 0$ only if $a = 0$ or $b = 0$), then $\neg a = 0$ for every $a \neq 0$.

A soft constraint [6] may be seen as a constraint where each instantiation of its variables has an associated preference. Given $S = (A, +, \times, 0, 1)$ and an ordered finite set of variables $V$ over a domain $D$, a soft constraint is a function that, given an assignment $\eta : V \rightarrow D$ of the variables, returns a value of the semiring, i.e., $c : (V \rightarrow D) \rightarrow A$. Let $\mathcal{C} = \{c : D^{\subseteq V} \rightarrow A\}$ be the set of all possible constraints that can be built starting from $S$, $D$ and $V$: any function in $\mathcal{C}$ depends on the assignment of only a (possibly empty) finite subset $I$ of $V$, called the support, or scope, of the constraint. For instance, a binary constraint $c_{x,y}$ (i.e., $\{x, y\} = I \subseteq V$) is defined on the support $\text{supp}(c) = \{x, y\}$. Note that $c\eta[v = d]$ means $c\eta'$ where $\eta'$ is $\eta$ modified with the assignment $v = d$. Note also that $c\eta$ is the application of a constraint function $c : (V \rightarrow D) \rightarrow A$ to a function $\eta : V \rightarrow D$: what we obtain is, thus, a semiring value $c\eta = a$.

\footnote{The boolean semiring can be used to represent classical crisp constraints.}

\footnote{The constraint function $\bar{a}$ always returns the value $a \in A$ for all assignments of domain values, e.g., the $0$ and $1$ functions always return $0$ and $1$ respectively.}
Given the set \( C \), the combination function \( \otimes : C \times C \rightarrow C \) is defined as
\[
(c_1 \otimes c_2) \eta = c_1 \eta \times c_2 \eta \quad [6];
\]
\( \text{supp}(c_1 \otimes c_2) = \text{supp}(c_1) \cup \text{supp}(c_2) \). Given the set \( C \),
the combination function \( \oplus : C \oplus C \rightarrow C \) is defined as
\[
(c_1 \oplus c_2) \eta = c_1 \eta + c_2 \eta \quad [5];
\]
\( \text{supp}(c_1 \oplus c_2) = \text{supp}(c_1) \cup \text{supp}(c_2) \). Informally, \( \otimes/\oplus \) builds a
new constraint that associates with each tuple of domain values for such variables a
semiring that is obtained by multiplying/summing the elements associated by
the original constraints to the appropriate sub-tuples. Given a constraint \( c \in C \)
and a variable \( v \in V \), the projection \( [6] \) of \( c \) over \( V \setminus \{v\} \), written \( c \downarrow_{V \setminus \{v\}} \) is
the constraint \( c' \) such that
\[
c' \eta = \sum_{d \in D} c_{\downarrow_{V \setminus \{v\}}} = c(v = d).
\]
Informally, projecting means computing the best possible rating over all values of the remaining variables.

The partial order \( \leq_S \) over \( C \) can be easily extended among constraints by
defining \( c_1 \leq_S c_2 \iff \forall \eta, c_1 \eta \leq_S c_2 \eta. \) In order to define constraint equivalence
we have \( c_1 \equiv_S c_2 \iff \forall \eta, c_1 \eta =_S c_2 \eta \) and \( \text{supp}(c_1) = \text{supp}(c_2) \).

In Fig. 1 we show a graphical example of four weighted constraints (i.e.,
defined in the weighted semiring), where we have \( c_3 \otimes c_4 = c_2, c_3 \subseteq c_4, c_2 \subseteq c_3, \)
\( c_1 \not\subseteq c_2 \) because of the gray region, where \( c_2 \subseteq c_1 \) instead; moreover,
in Fig. 1 we can see that
\[
\text{supp}(c_1) = \text{supp}(c_2) = \text{supp}(c_3) = \text{supp}(c_4) = \{x\}.
\]

An SCSP \([6]\) is defined as a quadruple \( P = (S, V, D, C) \),
where \( C \subseteq C \) is the constraint set of the problem \( P \). The best level of consistency notion defined as
blevel(P) = Sol(P) \not\subseteq \emptyset, \) where \( \text{Sol}(P) = \otimes C \) \([6]\). A problem \( P \) is \( \alpha \)-consistent
if \( \text{blevel}(P) = \alpha \) \([6]\); \( P \) is instead simply “consistent” iff \( \text{blevel}(P) >_S 0 \) \([6]\). \( P \)
is inconsistent if it is not consistent. Figure 2 shows an SCSP as a graph: \( S \)
corresponds to the weighted semiring, i.e., \( \langle \mathbb{R}^+ \cup \{+\infty\}, \min, +, \infty, 0 \rangle \). Variables
\( V = \{x, y\} \) and constraints \( C = \{c_1, c_2, c_3\} \) are represented respectively by
nodes and arcs (unary for \( c_1 \) and \( c_3 \), and binary for \( c_2 \)), and semiring values are
written to the right of each variable assignment of the constraint, where
\( D = \{a, b\} \). The solution of \( P \) in Fig. 2 associates a preference to every domain
value of \( x \) and \( y \) by combining all the constraints, i.e., \( \text{Sol}(P) = \otimes C \). For
instance, for the assignment \( \langle a, a \rangle \) (that is, \( x = y = a \)), we compute the sum of
1 (which is the value assigned to \( x = a \) in constraint \( c_1 \)), 5 (which is the value
assigned to \( x = a, y = a \) in \( c_2 \)) and 5 (which is the value for \( y = a \) in \( c_3 \)).
Hence, the resulting preference value for this assignment is 11. The blevel
for the example in Fig. 2 is 7, corresponding to the assignment \( x = a, y = b \).
4 Soft Constraint Automata

Constraint Automata were introduced in [2] as a formalism to describe the behavior and possible data flow in coordination models (e.g., the Reo language [2]); they can be considered as acceptors of Timed Data Streams (TDS) [1,2]. In this section we extend some of the definitions given in [2] in order to obtain Soft Constraint Automata (SCA).

We recall the definition of TDS from [1], while extending it using the softness notions described in Sec. 3: we name this result as Timed Weighted Data Streams (TWDS), which correspond to the languages recognized by SCA.5 For convenience, we consider only infinite behavior and infinite streams that correspond to infinite “runs” of our soft automata, omitting final states, including deadlocks.

Definition 1 (Timed Weighted Data Streams). Let Data be a data set, and for any set $X$, let $X^\omega$ denote the set of infinite sequences over $X$; given a semiring $S = \langle A, +, \times, 0, 1 \rangle$, a Timed Weighted Data Stream (TWDS) is a triplet:

$$\langle \lambda, l, a \rangle \in \text{Data}^\omega \times \mathbb{R}_+^\omega \times A^\omega \text{ such that, } \forall k \geq 0 : l(k) < l(k+1) \text{ and } \lim_{k \to +\infty} l(k) = +\infty$$

Thus, a TWDS triplet $\langle \lambda, l, a \rangle$ consists of a data stream $\lambda \in \text{Data}^\omega$, a time stream $l \in \mathbb{R}_+^\omega$ and a preference stream $a \in A^\omega$. The time stream $l$ indicates, for each data item $\lambda(k)$, the moment $l(k)$ at which it is exchanged (i.e., being input or output), while the $a(k)$ is a preference score for its related $\lambda(k)$.

CA [2] use a finite set $N$ of names, e.g., $N = \{n_1, \ldots, n_p\}$, where $n_i \ (i \in 1..p)$ is the $i$-th input/output port. The transitions of SCA are labeled with pairs consisting of a non-empty subset $N \subseteq N$ and a soft (instead of crisp as in [2]) data-constraint $c$. Soft data-constraints can be viewed as an association of data assignments with a preference for that assignment. Formally,

Definition 2 (Soft Data-Constraints). A soft data-constraint is a function $c : \{\{d_n | n \in N\} \to \text{Data}\} \to A$ defined over a semiring $S = \langle A, +, \times, 0, 1 \rangle$, where $\{d_n | n \in N\} \to \text{Data}$ is a function that associates a data item with every variable $d_n$ related to port name $n \in N \subseteq N$, and $\text{Data}$ is the domain of data items that pass through ports in $N$. The grammar of soft data-constraints is:

$$c_{\{d_n | n \in N\}} = \emptyset | 1 | c_1 \oplus c_2 | c_1 \otimes c_2 | \neg c$$

where $\{d_n | n \in N\}$ is the support of the constraint, i.e., the set of variables (related to port names) that determine its preference (see Sec. 3).

Informally, a soft data-constraint is a function that returns a preference value $a \in A$ given an assignment for the variables $\{d_n | n \in N\}$ in its support. In the sequel, we write $\text{SDC}(N, \text{Data})$, for a non-empty subset $N$ of $N$, to denote the set of soft data-constraints. We will use $\text{SDC}$ as an abbreviation for $\text{SDC}(N, \text{Data}).$

5 TWDSs do not imply time constraints, and thus our (soft) CA are not “timed” [2].
Note that in Def. 2 we assume a global data domain $Data$ for all names, but, alternatively, we can assign a data domain $Data_n$ for every variable $d_n$.

We state that an assignment $\eta$ for the variables $\{d_n \mid n \in N\}$ satisfies $c$ with a reference of $a \in A$ (see Sec. 3). Equivalence and implication for soft data-constraints are defined in Sec. 3: we respectively write $c_1 \equiv c_2$ and $c_1 \sqsubseteq c_2$.

Note that by using the boolean semiring (see Sec. 3), thus within the same semiring-based framework, we can exactly model the “crisp” data-constraints presented in the original definition of CA [2]. Therefore, CA are subsumed by Def. 3. Note also that weighted automata, with weights taken from a proper semiring, have already been defined in the literature [9]; in SCA, weights are determined by a constraint function instead.

**Definition 3 (Soft Constraint Automata).** A Soft Constraint Automaton over a domain $Data$, is a tuple $T_S = (Q, N, \rightarrow, Q_0, S)$ where i) $S$ is a semiring $\langle A, +, \times, 0, 1 \rangle$, ii) $Q$ is a finite set of states, iii) $N$ is a finite set of names, iv) $\rightarrow$ is a finite subset of $Q \times 2^N \times SDC \times Q$, called the transition relation of $T_S$, and v) $Q_0 \subseteq Q$ is the set of initial states. We write $q \xrightarrow{N,c,p} p$ instead of $(q, N, c, p) \in \rightarrow$. We call $N$ the name-set and $c$ the guard of the transition. For every transition $q \xrightarrow{N,c,p} p$ we require that i) $N \neq \emptyset$, and ii) $c \in SDC(N, Data)$ (see Def. 2). $T_S$ is called finite iff $Q \rightarrow$ and the underlying data-domain $Data$ are finite.

The intuitive meaning of an SCA $T_S$ as an operational model for service queries is similar to the interpretation of labeled transition systems as formal models for reactive systems. The states represent the configurations of a service. The transitions represent the possible one-step behavior, where the meaning of $q \xrightarrow{N,c,p} p$ is that, in configuration $q$, the ports in $n \in N$ have the possibility of performing I/O operations that satisfy the soft guard $c$ and that leads from configuration $q$ to $p$, while the other ports in $N \setminus N$ do not perform any I/O operation. Each assignment of variables $\{d_n \mid n \in N\}$ represents the data associated with ports in $N$, i.e., the data exchanged by the I/O operations through ports in $N$.

In Fig. 3 we show an example of a (deterministic) SCA. In Fig. 4 we define the weighted constraints $c_1$ and $c_2$ that describe the preference (e.g., a monetary cost) for the two transitions in Fig. 3, e.g., $c_1(d_L = 2) = 5$.

We now define $sdc_{T_S}(q, N, P)$, which is used in Sec. 5.1 for (bi)simulation:

**Definition 4.** For an SCA $T_S = (Q, N, \rightarrow, Q_0, S)$, a state $q \in Q$, $N \subseteq N'$, $P \subseteq Q$, and $\oplus$ that corresponds to the application of the $\oplus$ operator (see Sec. 3)
to all the constraints of a set (⊗ is commutative and associative), we define:

\[ sd_cT_S(q, N, P) = \bigoplus \{ c \mid q \xrightarrow{N,c} p \text{ for some } p \in P \} \]

Intuitively, \( sd_cT_S(q, N, P) \) is the weakest (i.e., with the best preference) soft data-constraint that ensures the existence of an \( N \)-transition from \( q \) to a state in \( P \). Note that \( sd_cT_S(q, N, P) = \emptyset \) if there is no \( N \)-transition from \( q \) to a \( P \)-state.

As introduced before, we define the language accepted by an SCA \( T_S \) as

\[ 1. \quad L(TWDS) = \bigcup_{q \in Q_0} L(TWDS(T_S, q)) \]

where \( L(TWDS) \) denotes the language accepted by the state \( q \) (viewed as the starting state) of \( T_S \). Considering, as an example, a two port automaton \( T_S \), the accepted languages on \( \mathcal{N} = \{L, M\} \) are defined as the set of all TWDS pairs \( \langle \langle \lambda, l, a \rangle, \langle \mu, m, b \rangle \rangle \) that have an infinite run in \( T_S \) starting in state \( q \). The data streams \( \lambda \) and \( \mu \) correspond to the data elements that flow through, respectively, \( L \) and \( M \); \( l \) and \( m \) contain the time instants at which these data flow operations take place. \( L(TWDS(T_S, q)) \) consists of all pairs \( \langle \langle \lambda, l, a \rangle, \langle \mu, m, b \rangle \rangle \) such that there exists a transition \( q \xrightarrow{N,c} q' \) that satisfies the following conditions (where we denote the tail of the stream \( \lambda = \lambda(0), \lambda(1), \lambda(2), \ldots \) as \( \lambda' = \lambda(1), \lambda(2), \ldots \)):

\[
\begin{align*}
&l(0) < m(0) \land N = \{L\} \land c(d_L = \lambda(0)) = a(0) \land \langle \langle \lambda', l', a' \rangle, \langle \mu, m, b \rangle \rangle \in L(TWDS(T_S, q)) \\
or &m(0) < l(0) \land N = \{M\} \land c(d_M = \mu(0)) = b(0) \land \langle \langle \lambda, l, a \rangle, \langle \mu', m', b' \rangle \rangle \in L(TWDS(T_S, q)) \\
or &l(0) = m(0) \land N = \{L, M\} \land c(d_L = \lambda(0), d_M = \mu(0)) = \langle a(0) = b(0) \land \langle \langle \lambda', l', a' \rangle, \langle \mu', m', b' \rangle \rangle \in L(TWDS) \}
\end{align*}
\]

where \( a(0), b(0) >_S 0 \). Although the above definition is circular in case \( q = q' \), a proper monotone operator can be formally defined [2]. As an example, the language accepted by the automaton in Fig. 3 equals the set \( \{ \langle \langle \lambda, l, a \rangle, \langle \mu, m, b \rangle \rangle \in TWDS \times TWDS \mid \forall k \geq 0 : \lambda(k), \mu(k) \in \mathbb{N} \land l(k) < m(k) < l(k + 1) \land a(k) = c_1(d_L = \lambda(k)), b(k) = c_2(d_M = \mu(k)) \} \), with \( a(k), b(k) \geq_S 0 \).

We now define the soft-join operator of two SCA, performing the (natural) join of two \( L(TWDS) \). We can use this operator to merge two queries (see Sec. 5).

**Definition 5 (Soft-Product Automaton (soft join)).** The soft-product automaton of two SCA \( T_S^1 = (Q_1, N_1, \longrightarrow_1, Q_{0,1}, S) \) and \( T_S^2 = (Q_2, N_2, \longrightarrow_2, Q_{0,2}, S) \) on the same semiring \( S \) is defined as \( T_S^3 = (Q_1 \times Q_2, N_1 \cup N_2, \longrightarrow, Q_{0,1} \times Q_{0,2}, S) \), where \( \longrightarrow \) is given by the following two rules (and the symmetric one for (2)):

\[
\begin{align*}
&1. \quad q_1 \xrightarrow{N_1,c_1} p_1, q_2 \xrightarrow{N_2,c_2} p_2, N_1 \cap N_2 = N_2 \cap N_1 \quad \langle q_1, q_2 \rangle \xrightarrow{N_1 \cup N_2, c_1 \oplus c_2} \langle p_1, p_2 \rangle \\
&2. \quad q_1 \xrightarrow{N_1,c_1} p_1, N \cap N_2 = \emptyset \quad \langle q_1, q_2 \rangle \xrightarrow{N} \langle p_1, q_2 \rangle
\end{align*}
\]
The first rule applies when there are two transitions in the automata that can fire together. This happens only if there is no shared name in the two automata that is present on one of the transitions, but not present on the other one. In this case, the transition in the resulting automaton is labelled with the union of the name sets on both transitions, and the data-constraint is the conjunction of the data-constraints of the two transitions. The second rule applies when a transition in one automaton can fire independently of the other automaton, which happens when the names on each transition are not included in the name set of the automaton of the other transition. The proof for correctness of the soft join derives from the correctness of the crisp join [2].

The hiding operator [2] abstracts the details of the internal communication in an SCA, and shows the observable external behaviour of a query. In SCA, the hiding operator \( ∃O[T] \) (see Def. 6) removes all information about names in \( O ⊆ N \), and removes the influence of the names in \( O \) from the SDC of \( T \): this operator removes \( O \) from the support of all soft constraints in \( T \).

Definition 6 (Hiding in Soft Constraint Automata). Let \( T_5 = (Q, N, →, Q_0, S) \) be an SCA and \( N, O ⊆ N \). The SCA \( ∃O[T] = (Q, N\setminus O, →_O, Q_0, O, S) \) is defined as follows. Let \( →^* \) be the transition relation such that \( q →^* p \) if there exists a finite path \( q \overset{O,c_1}{→} q_1 \overset{O,c_2}{→} q_2 \overset{O,c_3}{→} \cdots \overset{O,c_n}{→} q_n \), where \( q_0 = p \) and \( c_1, \ldots, c_n \) are satisfiable (i.e., \( c_i \neq \emptyset \)) and \( ∀c_i, \text{supp}(c_i) = O \). The set \( Q_0, O \) of an initial states is \( Q_0 \cup \{ p \in Q : q_0 →^* p \text{ for some } q_0 \in Q_0 \} \). The transition relation \( →_O \), where \( \downarrow \) is the soft constraint projection described in Sec. 3, is given by:

\[
\begin{align*}
q →^* p, &p \overset{N,c}{→} r, N' = N\setminus O \neq \emptyset, c' = c \downarrow \{d_n | n \in N'\} \\
q \overset{N',c'}{→}_O r
\end{align*}
\]

5 Expressing Preference-based Queries

In this section we use SCA (see Sec. 4) to model the queries we adopt to describe i) the behavior of the services a user is interested in, and ii) the preferences of the user with respect to data exchanged through I/O by the operations. The behavioral signature of a service is described via a (crisp) constraint automaton: the operations are described by the names on the transitions of the automaton, as described in Sec. 4; the ordering of the operations is enforced by the ordering of the reached states. Analogously, a query is described via an SCA, where we use SDC (see Def. 2) to model user’s preferences for the data used by the service operations. Matching these names with the actual names of the services in the database leads to a global preference for that service.

Our model assumes, ignoring details, the existence of a standard vocabulary (i.e., a domain-specific ontology) for messages and activities (e.g., OWL-S [15]). Therefore, we suppose that all names in the following examples are properly obtained from an ontology on services. Ontologies have already been used in the literature to help preference and similarity-based searches (see Sec. 2), and serve as a common dictionary for queries and services.
In Fig. 5 we show two first examples of soft queries: with $q_0$ the user looks for a bibliography-search service that is able to search for conference papers by $Author$, while in the case of $q_1$ the search is by $Title$. The user’s preferences on input data that these two services take are summarized by the two soft constraints $c_1$ and $c_2$, which are represented in Fig. 6. These examples can be modeled with the fuzzy semiring $\langle [0..1], \max, \min, 0, 1 \rangle$: $c_1$ states that the user prefers to have a search by first name (with a fuzzy score of 1), rather than to have it by full name (i.e., 0.8) or by last name (i.e., 0.2): $c_2$ states that the user prefers to have a search using the conference title instead of the paper title. The preference is equal to the bottom value of the semiring where not stated (here, $0 = 0$). A possible scenario for this example corresponds to a situation where the user remembers the first name of the author, or the conference where they met, but he has a vague memory of the author’s last name, and of the title of the presented paper.

Suppose now that our database contains the four services/operations represented in Fig. 7. According to the preferences expressed by $c_1$ and $c_2$ in Fig. 6, queries $q_0$ and $q_1$ in Fig. 5 return different preferences for each operation, depending on the instantiation of variables $d_{Author}$ and $d_{Title}$. Considering $q_0$, operations $a$, $b$, and $d$ have respective preferences of 0.2, 1, and 0.8. If query $q_1$ is used instead, the possible results are operations $c$ and $d$, with respective preferences of 1 and 0.3. When more than one service is returned as the result of a search, the end user has the freedom to choose the best one according to his preferences: for the first query $q_0$, he can opt for service $b$, which corresponds to a preference of 1 (i.e., the top preference), while for query $q_1$ he can opt for $c$ (top preference as well). A possible programming-like declaration for opera-

![Fig. 5. Two soft Constraint Automata](image1)

![Fig. 6. The definition of $c_1$ and $c_2$ in representing two different queries.](image2)

![Fig. 7. A database of services for the query in Fig. 5; $d$ performs both kinds of search.](image3)
tion a in Fig. 7 is “void Author(last)”. Note that a fifth possible service in the
database may implement a search by name initials, but according to $c_1$ in Fig. 6,
the user’s preference for this service would be 0, i.e., $c(d_{Author} = \text{Initials}) = 0$.

Note also that we can define $n$-ary soft constraints for more than one input
data at the same time, in order to relate the preference for the values of two or
more I/O data. For example, if we want to search by author and title at the same
time, we can add a binary constraint $c$ on \{d$_{Author}$, d$_{Title}$\}, such that $c(d_{Author} = \text{First}, d_{Title} = \text{Conference}) = a_1$, $c(d_{Author} = \text{First}, d_{Title} = \text{Paper}) = a_2$, and
$a_1 > S a_2$. This means that, when we know the first name of the author, we
prefer to search using the title of the conference, instead of the paper title.

In Fig. 8 we provide a more complex example of a soft query, where we show
a classical on-line purchase scenario cited in Sec. 1, considering its behavior.
In this case, the requirements of the user are
i) a login/logout service,
ii) an
electronic basket that can be filled with the user’s orders (at least one item has
to be added before proceeding further),
iii) a decision on the shipping method
and, finally, iv) a payment service. Therefore, this query is expressed with the
help of four different states modeling its behavior. The $SDC$ expressing the user’s
preferences are represented in Fig. 9, where we suppose that the user expresses
no preference for the data concerning the Logout service, i.e., $c_2 = 1$. Note that,
after the payment, the user can make successive purchases without logging out.

In the following we show that the join and hiding operators presented in
Sec. 4 can be used to operate on queries. Note that, by using the same view
presented in [14], we can slightly modify the join operator in Sec. 4 in order to
be able to compose the two queries in Fig. 10 into the one presented in Fig. 8.
Query composition is useful in order to reuse parts of a query in another one, or
to split the query into different knowledge domains. For example, in Fig. 10 the
first query can be decided by the internal IT department of the company that

![Diagram](image)

**Fig. 8.** A more complex soft query for the on-line purchase scenario presented in Sec. 1.

![Diagram](image)

**Fig. 9.** The definition of $c_1$, $c_3$, $c_4$, $c_5$ for the query in Fig. 8 (we suppose $c_2 = 1$).
needs to use the service, according to their internal security regulations, whereas the second query can be decided by the purchasing department of the company.

For this, the join operator has to be slightly modified, as defined in [14] for the composition of services.

The hiding operator presented in Sec. 4 can be used to hide some over-constraining information from the query, if, for instance, the previous search has led to a “no result” answer for the user. Suppose we ask query $q_0$ in Fig. 11 having a database as the one represented in Fig. 7. In this case, no result is returned because no service implements a search by Author and Title at the same time. A user may relax the query by hiding Title, and ask again, this time obtaining as possible response services $a$, $b$ and $d$ in Fig. 7.

5.1 Formally Reasoning on the Similarity of Queries

In the sequel we adapt the notions of bisimulation and simulation on (crisp) constraint automata [2] for SCA by considering soft constraints, instead of crisp ones.

**Definition 7 (Soft Bisimulation).** Let $\mathcal{T}_S = (Q, N, \rightarrow, Q_0, S)$ be an SCA and let $\mathcal{R}$ be an equivalence relation on $Q$. $\mathcal{R}$ is called a soft bisimulation for $\mathcal{T}_S$ if, for all pairs $(q_1, q_2) \in \mathcal{R}$, all $\mathcal{R}$-equivalence classes $P \in \mathbb{Q}/\mathcal{R}$, and every $N \subseteq \mathbb{N}$: $sdc_{\mathcal{T}_S}(q_1, N, P) \equiv sdc_{\mathcal{T}_S}(q_2, N, P)$.

Recall that $c_1 \equiv c_2$ iff $c_1 \eta = c_2 \eta = a$ for every assignment $\eta$ (see Sec. 3). States $q_1$ and $q_2$ are called bisimulation-equivalent ($q_1 \sim q_2$) iff there exists a bisimulation $\mathcal{R}$ with $(q_1, q_2) \in \mathcal{R}$. Two automata $\mathcal{T}_S^1$ and $\mathcal{T}_S^2$ are bisimulation-equivalent ($\mathcal{T}_S^1 \sim \mathcal{T}_S^2$) iff their initial states are bisimulation-equivalent [2].

**Definition 8 (Soft Simulation).** Let $\mathcal{T}_S = (Q, N, \rightarrow, Q_0, S)$ be an SCA and let $\mathcal{R}$ be a binary relation on $Q$. $\mathcal{R}$ is called a soft simulation for $\mathcal{T}_S$ if, for all pairs $(q_1, q_2) \in \mathcal{R}$, all $\mathcal{R}$-upward closed sets $P \subseteq Q$, and every $N \subseteq \mathbb{N}$: $sdc_{\mathcal{T}_S}(q_1, N, P) \subseteq sdc_{\mathcal{T}_S}(q_2, N, P)$.
An automaton \( T_1 \) simulates another automaton \( T_2 \) iff every initial state of \( T_1 \) is simulated by an initial state of \( T_2 \); this relationship is denoted as \( T_1 \preceq T_2 \).

Soft bisimulation can be seen as a method to check the equivalence of two \( \mathcal{L}_{TWDS} \) languages, while soft simulation concerns language inclusion, as explained in [2] for the crisp version of CA. Moreover, since our timed streams are weighted as explained in Def. 1 we can prove the following proposition:

**Proposition 1.** Given two soft constraint automata \( T_1 \) and \( T_2 \) able to parse the languages \( L_{TWDS_1} \) and \( L_{TWDS_2} \), respectively, for each stream \( \langle \lambda, l, a \rangle \in TWDS_1 \) there exists a stream \( \langle \mu, m, b \rangle \in TWDS_2 \) such that:

- If \( T_1 \sim T_2 \) and \( \forall k, \lambda(k) = \mu(k) \), then \( a(k) =_S b(k) \).
- If \( T_1 \preceq T_2 \) and \( \forall k, \lambda(k) = \mu(k) \), then \( a(k) \geq_S b(k) \).

The proof derives from the fact that, if \( T_1 \sim T_2 \), then \( sdc_{T_1}(q_1, N, P) \equiv sdc_{T_2}(q_1, N, P) \), and, if \( T_1 \preceq T_2 \), \( sdc_{T_1}(q_1, N, P) \subseteq sdc_{T_2}(q_2, N, P) \).

Bisimulation can be used to check if two queries are equivalent, that is, if they search the same services with the same preferences expressed by a user. Moreover, a simulation between \( T_1 \preceq T_2 \) can be used to check if the query expressed through \( T_1 \) is entailed by \( T_2 \), and, consequently, the latter’s returned services are a subset of the former’s. Note that simulation/bisimulation for weighted automata (but not for SCA) has already been defined in the literature [9].

### 6 Similarity-based Queries

In Sec. 5 we adopted the theory presented in Sec. 4 to represent the queries using crisp synchronization constraints (i.e., “crisp names”) and soft data-constraints. This way, it is possible to guide the search according to the preferences of the user concerning the data used by the service operations.

In this section, we focus on similarity-based search, instead of preference-based search: transition names are not crisp anymore, allowing for different operations considered somehow similar for the purposes of a user’s query. Note that a similar service can be used, e.g., when the “preferred” one is down due to a fault, or when it offers bad performances, e.g., due to the high number of requests. Definition 9 formalizes the notion of soft synchronization-constraint.

**Definition 9 (Soft Synchronization-constraint).** A soft synchronization-constraint is a function \( c : (V \to \mathbb{N}) \to A \) defined over a semiring \( S = \langle A, +, \times, 0, 1 \rangle \), where \( V \) is a finite set of variables for each I/O ports, and \( N \) is the set of I/O port names of the SCA.

For example, suppose that a user asks only query \( q_0 \) in Fig. 5. The possible results are services \( a, b \) and \( d \) in the database of Fig. 7, since service \( c \) performs a search based on the Title of the paper only, and not on the Author. However, the two services are very similar, and a user may be satisfied also by retrieving (and then, using) service \( c \). This can be accomplished with the query in Fig. 12,
where \( c_x(x = \text{Author}) = 1 \), and \( c_x(x = \text{Title}) = 0.7 \). In Fig. 13, we show a similarity-based query for our on-line purchase scenario: in this case, we have \( V = \{x_1, x_2, x_3, x_4, x_5, x_6\} \), and the domain for each of these variables is \( N = \{\text{MutualLogin}, \text{Login}, \text{Logout}, \text{AddToBasket}, \text{Shipping}, \text{Charging}, \text{SendEstimate}\} \).

A user can also choose for a mutual login in the first step, and for an estimate of the price instead of a direct charging service.

### 6.1 A Mapping to Soft Constraint Satisfaction Problems

In this section we map our similarity-based search into an SCSP \( P \) (see Sec. 3); by solving \( P \), we find the services in the database closest to the requirements expressed by a user’s query. We use this solving method for two fundamental reasons: first, constraint programming is a declarative and very expressive means to quickly define the acceptable results in terms of constraint relationships. Second, SCSPs (and in general, Constraint Programming) come with several AI-based techniques that can be used to cut the search space and improve efficiency. For example, an \( \alpha \)-cut on the branch-and-bound search can be used to stop the search as soon as the preference of the partial solution becomes worse than a threshold specified by the user. In this way, we can find only the \( \alpha \)-consistent solutions (see Sec. 3), thus sparing computational resources for those solutions with a worse-than-\( \alpha \) preference, which would not be selected by the user after all.

This is particularly desirable with very large databases of services, containing interfaces with thousands of operations and thousands of behavioral states.

**Mapping.** We propose a mapping \( M \) such that, given the SCA \( T_S = (Q, N, \rightarrow, Q_0, S) \) (i.e., our query), and a database of services \( DB = \{T_1, T_2, \ldots, T_h\} \) represented by a crisp constraint automata [2], we obtain \( M(T_S, DB) = P \), where \( P = (S, V, D, C) \) is an SCSP (defined in Sec. 3). For each transition \( i \) in the automaton \( T_S \) modeling the query, we use two variables \( x_i, y_i \) representing the source and destination states of the transition. A variable \( z_i \) represents the operation names that we associate with transition \( i \). Therefore, \( V = \bigcup \{x_i, y_i, z_i\} \), for \( i = 1 \ldots \#\text{transitions}(T_S) \). The domain of the state variables in \( V \) is \( Q_1 \cup Q_2 \cup \ldots \cup Q_h = Q_{DB} \) (where \( Q_h \) is the set of states in automaton \( T_h \)), and the domain of the operation names in \( V \), \( N_{DB} \), is the set of operation names used by all services in \( DB \). We identify two different classes of constraints to build the \( C \) component of \( P \):
Automaton-structure constraints. With this set of constraints, we force the solutions (i.e., the crisp automata in $DB$) to respect the structure of automaton $T_5$. For each $q_a N_i q_b$ in some $T_5$ in $DB$ ($q_a, q_b \in Q_{DB}$), we have $c_{x_1,y_1}(x_i = q_a, y_i = q_b) = 1$, and $0$ otherwise. In addition, we also need to enforce the sequence of the states/transitions in the solution, according to the one expressed by the query. For example, if we have $q_{x_1} \rightarrow q_{y_1}$ and $q_{x_2} \rightarrow q_{y_2}$, and $y_i = x_j$ in $T_5$, we need to add $c_{x_1,y_1}(x_j = y_i) = 1$, and $0$ otherwise.

Name-preference constraints. These constraints model the preference for the names on the transitions. For each $q_a N_i q_b \in DB$, $c_{x_1,y_1,z_1}(x_i = q_a, y_i = q_b, z_i = n_i) = a$, where $a \in A$ (the set of preferences in $S$) represents the preference for the name $n_1$.

Example 1. Here we list all the automaton-structure constraints with a preference better than $0$, that model the query in Fig 13: $c_{x_1,y_1}(x_1 = q_a, y_1 = q_b) = 1$, $c_{x_2,y_2}(x_2 = q_a, y_2 = q_d) = 1$, $c_{x_1,y_2}(x_1 = y_2) = 1$, $c_{x_2,y_1}(x_2 = y_1) = 1$, $c_{x_3,y_3}(x_3 = q_a, y_3 = q_b) = 1$, $c_{x_3,y_2}(x_3 = y_2) = 1$, $c_{x_4,y_4}(x_4 = q_a, y_4 = q_b) = 1$, $c_{x_2,y_4}(x_2 = y_4) = 1$, $c_{x_4,y_2}(x_4 = y_2) = 1$, $c_{x_5,y_5}(x_5 = q_a, y_5 = q_b) = 1$, $c_{x_3,y_3}(x_3 = y_3) = 1$, $c_{x_6,y_6}(x_6 = q_a, y_6 = q_b) = 1$, $c_{x_6,y_6}(x_6 = y_6) = 1$, $c_{x_3,y_3}(x_3 = y_6) = 1$, for each $q_a N_i q_b$ in the service database.

In the following we list all name-preference constraints with a preference better than $0$: $c_{x_1,y_1,z_1}(x_1 = q_a, y_1 = q_b, z_1 = \{Login\}) = 0.8$, $c_{x_1,y_1,z_1}(x_1 = q_a, y_1 = q_b, z_1 = \{MutualLogin\}) = 1$, $c_{x_2,y_2,z_2}(x_2 = q_a, y_2 = q_b, z_2 = \{Logout\}) = 1$, $c_{x_3,y_3,z_3}(x_3 = q_a, y_3 = q_b, z_3 = \{AddToBasket\}) = 1$, $c_{x_4,y_4,z_4}(x_4 = q_a, y_4 = q_b, z_4 = \{AddToBasket\}) = 1$, $c_{x_5,y_5,z_5}(x_5 = q_a, y_5 = q_b, z_5 = \{Shipping\}) = 1$, $c_{x_6,y_6,z_6}(x_6 = q_a, y_6 = q_b, z_6 = \{Charging\}) = 1$ and $c_{x_6,y_6,z_6}(x_6 = q_a, y_6 = q_b, z_6 = \{SendEstimate\}) = 0.5$, for each $q_a N_i q_b$ in the service database.

7 Conclusion

In this paper, we have proposed a general formal framework to express both preference and similarity-based queries for the SOC paradigm. This framework has evolved from Constraint Automata [2] (to model the behavior of services) and semiring-based soft constraints [6,5] (to model preference values). We have merged these two ingredients to obtain SCA, which comprise the formalism we use to model these kinds of queries. The resulting framework can parametrically deal with different systems of semiring-based preferences.

In the future, we will automate the search by implementing the mapping proposed in Sec. 6.1 using a real constraint programming environment, such as CHOCO [13], and test its performance results. Moreover, we want to unify both preference and similarity-based queries in a single framework, to be able to express similarity-based queries with preferences on I/O data.

References

Reconfiguration mechanisms for service coordination*

Nuno Oliveira and Luis S. Barbosa
HASLab/INESC TEC & Universidade do Minho
{nunooliveira,lsb}@di.uminho.pt

Abstract. Models for exogenous coordination provide powerful glue-code, in the form of software connectors, to express interaction protocols between services in distributed applications. Connector reconfiguration mechanisms play, in this setting, a major role to deal with change and adaptation of interaction protocols. This paper introduces a model for connector reconfiguration, based on a collection of primitives as well as a language to specify connectors and their reconfigurations.

1 Introduction

The purpose of a service-oriented architecture (SOA)[5,6] is to address requirements of loosely coupled and protocol-independent distributed systems, where software resources are packaged as self-contained services providing well-defined functionality through publicly available interfaces. The architecture describes their interaction, ensuring, at the same time, that each of them executes independently of the context or internal state of the others.

Over the years a multitude of technologies and standards [1] have been proposed for describing and orchestrating web services, publish and discover their interfaces and enforce certain levels of security and QoS parameters. Either to respond to sudden and significative changes in context or performance levels, or simply to adapt to evolving requirements, some degree of adaptability or reconfigurability is typically required from a service-oriented architecture. By a (dynamic) reconfiguration we mean a process of adapting the architectural current configuration, once the system is deployed and without stopping it [9], so that it may evolve according to some (emergent) requirements [7] or change of context.

Reconfigurations applied to a SOA may be regarded from two different point of views. From one of them, they target individual services [16]. In particular, such reconfigurations are concerned with dynamic update of services, substitution of a service by another with compatible interfaces (but not necessarily the

* This work is funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-010047. The first author is also supported by an Individual Doctoral Grant with reference number SFRH/BD/71475/2010
same behaviour) or even their plain removal. Such reconfigurations are usually triggered by external stimulus [14,8,12,13,18]. Form another point of view, a reconfiguration is entirely decided by the system itself and targets the way components or services interact with each other, as well as the internal QoS levels measured along such interactions. In particular, such reconfigurations deal with substitution, addition or removal of communications channels, moving communication interfaces from a service to another or rearranging a complex interaction structure.

This paper studies reconfiguration mechanisms for the service interaction layer in SOA. Adopting a coordination-based view of interaction [15], the model proposed here represents the 'gluing-code' by a graph of channels whose nodes represent interaction points and edges are labelled with channel identifiers and types. A channel abstracts a point-to-point communication device with a unique identifier, a specified behaviour and two ends. It allows for data flow by accepting data on a source end and dispensing it from a sink end. We call such a graph a coordination pattern. A subset of its nodes are intended to be plugged to concrete services, forming the pattern interface.

To keep things concrete, we assume channels in a coordination pattern are described in a specific coordination model, that of Reo [3,2]. Actually, this choice is not essential: the reconfiguration mechanisms are directly defined over the graph and concern only its topology. Only when one intends to reason about the system’s behaviour or compare the behavioural effect of a reconfiguration, does the specific semantics of the underlying coordination model become relevant. Such is not addressed, however, in this paper.

Coordination patterns are introduced in Section 4 and instantiated in the context of the Reo coordination model. Section 3 discusses reconfigurations, formally defining a collection of primitives. It is shown how the latter can be combined to yield ‘big-step’ reconfiguration patterns which manipulate significative parts of a pattern structure. The CooPLa language is introduced in Section 4 as an executable notation for specifying both coordination and reconfiguration patterns. Reconfiguration mechanisms are illustrated through a detailed example in Section 5. Section 6 concludes the paper.

2 Coordination patterns

A pattern is an effective, easy to learn, repeatable and proven method that may be applied recurrently to solve common problems [5]. They are common in several domains of Software Engineering, namely in SOA [17] and business process [20].

Similarly, in this paper, a coordination pattern encodes a reusable solution for an architectural (coordination) problem in the form of a specific sort of interaction between the system constituents. A solution for an architectural problem is, therefore, the description of interaction properly designed to meet a set of requirements or constraints. It is reflected in a coordination protocol,
which acts as glue-code for the components or services interacting within the system.

Formally, a coordination pattern is presented by a graph of channels whose nodes represent interaction points and edges are labelled with channel identifiers and types. As explained in the Introduction, we adopt here the Reo framework [2], in order to give a concrete illustration of our approach.

Let Name and Node denote, respectively, a set of unique names and a set of nodes associated either with coordination patterns or channels. A node can also be seen as an interaction port. It is assumed the following set of primitive types of channels (see Fig. 1) with the usual Reo [3,2] semantics.

$$\mathcal{T}_{\text{ype}} \overset{\text{def}}{=} \{\text{sync}, \text{lossy}, \text{fifo}_s, \text{fifo}_e, \text{drain}\}$$

Each channel has exactly two ends and are, normally, directed (with a source and a sink end) but Reo also accepts undirected channels (i.e., channels with two ends of the same sort). Channel ends form the nodes of coordination patterns. A node may be of three distinct types: (i) source node, if it connects only source channel ends; (ii) sink node, if it connects only sink channel ends and (iii) mixed node, if it connects both source and sink nodes. Fig. 1 recalls the basic channels used in Reo through the composition of which complex coordination schemes can be defined. The sync channel transmits data from one end to another whenever there is a request at both ends synchronously, otherwise one request shall wait for the other. The lossy channel behaves likewise, but data may be lost whenever a request at the source end is not matched by another one at the sink end. Differently, a fifo channel has a buffering capacity of one memory positions, therefore allowing for asynchronous occurrence of I/O requests. Qualifier e or f refers to the channel internal state (either empty or full). Finally, the synchronous drain channel accepts data synchronously at both ends and loses it.

We use $\mathcal{P}$ to denote the set of all coordination patterns. A coordination pattern is defined as follows:

**Definition 1 (Coordination pattern).** A coordination pattern is a triple

$$\rho \overset{\text{def}}{=} (I, O, R)$$

- $R \subseteq \text{Node} \times \text{Name} \times \mathcal{T}_{\text{ype}} \times \text{Node}$ is a graph on connector ends whose edges are labelled with instances of primitive channels, denoted by a channel identifier (of type Name) and a type (of type $\mathcal{T}_{\text{ype}}$);
- $I, O \subseteq \text{Node}$ are the sets of source and sink ends in graph $R$, corresponding to the set of input and output ports in the coordination pattern, respectively.
Clearly, every channel instance gives rise to a coordination pattern. For example pattern
\[
\rho_s = \{\{a\}, \{b\}, \{(a, sc, sync, b)\}\}
\]
corresponds to a single synchronous channel, identified by \(sc\), linking an input port \(a\) to an output port \(b\). Similarly, plugging to its output port two lossy channels yields a lossy broadcaster which replicates data arriving at \(a\), if there exist pending reading requests at \(d\) and \(e\):
\[
\rho_b = \{\{a\}, \{d, e\}, \{(a, sc, sync, b)\}, \{(b, l_1, lossy, d)\}, \{(b, l_2, lossy, e)\}\}
\]
A drain, on the other hand, has two source, but no sink, ends. Therefore, a pattern formed by an instance of a drain channel resorts to a special end \(\perp\in Node\) which intuitively represents absence of data flow. Thus, and for example,
\[
\rho_d = \{\{a, b\}, \emptyset, \{(a, ds, drain, \perp)\}, \{(b, ds, drain, \perp)\}\}
\]
As a matter of fact, well-formedness invariants of coordination patterns are required. For instance (i) the \(\perp\) ports can never be connected to other ports
(ii) a name may only be associated to two different ports and a unique channel type (notice the veracity of this also in the drain example) or (iii) only a single channel is allowed to connect two consecutive nodes. We assume the existence of such invariants, and do not address them in this paper.

3 Architectural reconfigurations

This section discusses reconfigurations of coordination patterns. We take a rather broad view of what a reconfiguration is: any transformation obtained through a sequence of elementary operations, described below, is qualified as a reconfiguration. Our aim is to build a framework in which such transformations can be defined and the effect of their application to a specific pattern assessed. Later, one may restrict this set, for example by ruling out transformations which do not preserve the pattern input-output interface or fail to lead to patterns with a behaviour which simulates (or bisimulates) the original one. Such considerations, however, require the assumption of a specific semantics for coordination patterns, easily built from any \(\text{Reo}\) semantic model, but which lies outside the more ‘syntactic’ scope of this paper.

Definition 2 (Reconfiguration). A reconfiguration is a sequence \(r\) of operations \(\langle o_0, o_1, \ldots, o_n \rangle\), where each \(o_i\) belongs to the set
\[
Op \overset{\text{def}}{=} \{\text{par, join, split, remove}\}
\]
of elementary reconfigurations, specified below. The application of a reconfiguration \(r\) to a pattern \(\rho\) yields a new pattern and is denoted by \(\rho \bullet r\).
3.1 Primitive reconfigurations

Let us start by defining the set of elementary reconfigurations of a coordination pattern. The simplest reconfiguration is *juxtaposition*. Intuitively, it sets two coordination patterns in parallel without creating any connection between them. Formally,

**Definition 3 (The \( \text{par} \) operation).** Let \( \rho_1 = (I_1, O_1, R_1) \) and \( \rho_2 = (I_2, O_2, R_2) \) be two coordination patterns. Then,

\[
\rho_1 \cdot \text{par}(\rho_2) = (I_1 \uplus I_2, O_1 \uplus O_2, R_1 \uplus R_2)
\]

where \( \uplus \) is set disjoint union.

The \( \text{par} \) operation assumes disjunction of nodes and channel identifiers in the patterns to be joined. This is assumed without loss of generality, because formally a disjoint union of all identifiers is previously made.

The second elementary reconfiguration is *join*. Intuitively, it creates a new node \( j \) that superposes all nodes in a given set \( P \). This operation adds fresh node \( j \) as a new input or output port if all the nodes in \( P \) are, respectively, input or output ports in \( \rho \). Formally,

**Definition 4 (The \( \text{join} \) operation).** Let \( \rho = (I, O, R) \in \mathcal{P} \), \( P \subseteq \mathcal{N}od \) and \( j \in \mathcal{N}od \). Then,

\[
\rho \cdot \text{join}(P, j) = (I', O', R')
\]

- \( R' = 2^{j^{\text{node}}}(R) \), with
  \[
  \text{join}_{P,j}(q, id, t, s) = ((p \in P \rightarrow j, p), id, t, (s \in P \rightarrow j, s))
  \]
- \( I' = (P \subseteq I \rightarrow \{j\}, \emptyset) \cup (I \setminus P) \)
- \( O' = (P \subseteq O \rightarrow \{j\}, \emptyset) \cup (O \setminus P) \)

The notation \((\phi \rightarrow s, t)\) corresponds to McCarthy’s conditional, returning \( s \) or \( t \) if predicate \( \phi \) is true or false, respectively. Also note that the power set of a set \( A \) is denoted by \( 2^A \) and, for a function \( f \) from \( A \) to \( B \), \( 2^f(X) = \{ f x \mid x \in X \} \).

The join operation has two pre-conditions. Clearly, node \( j \) must be a fresh name in \( \rho \). Additionally, every node in \( P \) shall exist as a node of \( \rho \). Formally, \( \forall p \in P, \exists e \in R, \pi_1(e) = p \lor \pi_4(e) = p \).

The dual to join is the \( \text{split} \) operation which takes a node \( p \) in a pattern and breaks connections, separating all channel ends coincident in \( p \). Technically this is achieved by renaming every occurrence of node \( p \) in all \( e \in R \) to a fresh name \( a.p \) or \( p.a \) depending on whether \( p \) appears as a sink node in \( e \) and \( a \) is the corresponding source end, or the other way round. Thus,

**Definition 5 (The \( \text{split} \) operation).** Let \( \rho = (I, O, R) \in \mathcal{P} \), and \( p \in \mathcal{N}od \). Then,

\[
\rho \cdot \text{split}(p) = (I', O', R')
\]
- $R' = 2^{sp_R}(R)$, with
  $$sp_R(q, ch, t, s) = \langle ((q = p) \rightarrow p.s, q), ch, ((s = p) \rightarrow q.p, s) \rangle$$

- $I' = (I \setminus \{p\}) \cup \{ p.x \mid e \in R' \land \pi_1(e) = p.x \}$
- $O' = (O \setminus \{p\}) \cup \{ x.p \mid e \in R' \land \pi_4(e) = x.p \}$

Finally, the remove operation removes a channel from a coordination pattern, if it exists.

**Definition 6 (The remove operation).** Let $\rho = \langle I, O, R \rangle \in \mathcal{P}$, and $ch \in \text{Name}$. Then,

$$\rho \bullet \text{remove}(ch) = \langle I', O', R' \rangle$$

where, for $L = \{ e \mid e \in R \land \pi_2(e) = ch \}$ and $L' = \{ \pi_1(e), \pi_4(e) \mid e \in L \}$

- $R' = R \setminus L$
- $I' = (I \setminus L') \cup \{ y \mid \langle x, ch, t, y \rangle \in L \land y \neq \bot \land \text{in}(y, R') \}$
- $O' = (O \setminus L') \cup \{ x \mid \langle x, ch, t, y \rangle \in L \land x \neq \bot \land \text{out}(x, R') \}$

where

$$\text{in}(x, S) \triangleq (\exists e \in S. \pi_4(e) = x) \land (\exists e \in S. \pi_1(e) = x)$$

$$\text{out}(x, S) \triangleq (\exists e \in S. \pi_1(e) = x) \land (\exists e \in S. \pi_4(e) = x)$$

Intuitively, removing a channel corresponds to deleting all the ends of that channel from the pattern interface. This may create free nodes that must be added to the coordination pattern interface. The second member of the union in the $I'$ or $O'$ definition, formally encodes these cases.

### 3.2 Reconfiguration patterns

Practice and experience in software architecture inspire the definition of patterns for reconfiguring architectures. As stated in the Introduction, the focus of traditional reconfiguration is set on the replacement of individual components, rather than on the interaction protocols. Our patterns, on the other hand, focused on the latter, but still at this lower-level we are interested in defining ‘big step’ reconfigurations, replacing simultaneously significant parts of a pattern.

Fig. 2 sums up the set of such reconfiguration patterns we have found useful in practice. The first one removes from a pattern a whole set of channels, applying the remove primitive systematically,

**Definition 7 (The removeP pattern).** Let $\rho \in \mathcal{P}$ and $Cs$ be a set of channels to remove. Then,

$$\rho \bullet \text{removeP}(Cs) = rS(\rho, Cs)$$

where

$$rS(\rho, \emptyset) = \rho$$

$$rS(\rho, Cs) = \text{let } c \in Cs \text{ in } rS(\rho \bullet \text{remove}(c), Cs \setminus \{c\})$$
Another common reconfiguration overlaps two patterns by joining nodes from both of them. This is specified by a set of triples indicating which nodes are to be overlapped and a fresh name for the result. Formally,

**Definition 8 (The overlapP pattern).** Let $\rho, \rho_r \in \mathcal{P}$ and $X$ be a set of triples of nodes, where the first component is a node of $\rho$, the second one is a node of $\rho_r$, and the third is a fresh node in both coordination patterns. Then,

$$\rho \ast \text{overlapP} (\rho_r, X) = rO(\rho \ast \text{par}(\rho_r), X)$$

where

$$rO(\rho, \emptyset) = \rho$$

$$rO(\rho, X) = \begin{cases} \text{let } e_i \in X, E_i = \{\pi_1(e_i), \pi_2(e_i)\}, & \text{in } rO(\rho \ast \text{join}(E_i, \pi_3(e_i)), X \setminus \{e_i\}) \\
\end{cases}$$

The insertP pattern puts both patterns side by side, uses split to make room for a new pattern to be added, as shown in Fig. 2, and join to re-build connections. Formally,

**Definition 9 (The insertP pattern).** Let $\rho, \rho_r \in \mathcal{P}$ and $n, m_i, m_o, j_1, j_2 \in \text{Node}$, where $n$ is a node of $\rho$, $m_i, m_o$ are input and output nodes, respectively, of $\rho_r$ and $j_1, j_2$ are fresh nodes. Then,

$$\rho \ast \text{insertP}(\rho_r, n, m_i, m_o, j_1, j_2) = \begin{cases} \rho_1 = \rho \ast \text{par}(\rho_r) & \\
\rho_2 = \rho_1 \ast \text{split}(n) & \\
I_{sp} = \pi_1(\rho_2) \setminus \pi_1(\rho_1) & \\
O_{sp} = \pi_2(\rho_2) \setminus \pi_1(\rho_1) & \\
\rho_3 = \rho_2 \ast \text{join}(O_{sp} \cup \{m_i\}, j_1) & \\
in \rho_3 \ast \text{join}(I_{sp} \cup \{m_o\}, j_2) & \\
\end{cases}$$

Replacing a sub-pattern involves removing the old structure followed by the overlap of the new pattern. A key operation is $\text{remNodes}$ which computes the nodes to be removed.

**Definition 10 (The replaceP pattern).** Let $\rho, \rho_r \in \mathcal{P}$ and $X$ be the set of triples of nodes, where the first component is a node of $\rho$, the second one is a node of $\rho_r$ and the third is a fresh node in both coordination patterns. Then,

$$\rho \ast \text{replaceP}(\rho_r, X) = (\rho \ast \text{removeP}(\pi_2(\text{remNodes}(\rho, 2^{\pi_1}(X)))) \ast \text{overlapP}(\rho_r, X))$$
Finally, the **implodeP** pattern collapses a set of nodes taken as the interface of a sub-structure. Formally,

**Definition 11 (The implodeP pattern).** Let $\rho = \langle I, O, R \rangle \in \mathcal{P}$, $j$ be a fresh node in $\rho$ and $X \subseteq \text{Node}$ be a set of nodes of $\rho$, which represent the border of the structure to implode. Then,

$$\rho \cdot \text{implodeP}(X, j) = \text{let} (Ch, N) = \text{remNodes}(\rho, X)$$

$$\rho_1 = \rho \cdot \text{removeP}(Ch)$$

$$M = \text{updateNodes}(N)$$

$$\text{in } \rho_1 \cdot \text{join}(M, j)$$

where $\text{updateNodes}(N) = N \cap \{ x | x \neq \perp \land (x = \pi_1(e) \lor x = \pi_4(e)) \land e \in R \}$

Operation $\text{remNodes}$, used above, is defined as follows:

$$\text{remNodes} : \mathcal{P} \times \text{Node} \times \text{Node} \to \text{Name} \times \text{Node}$$

$$\text{remNodes}(\langle I, O, R \rangle, N) =$$

let $N_1 = N \cup \{ x \not\in N | ((x, id, t, \perp), (a, id, t, \perp) \in R) \lor ((\perp, id, t, x), (\perp, id, t, a) \in R) \land a \in N \}$

$N_2 = \bigcup_{n \in N_1} \pi_1(\text{collNodes}(n, \emptyset, \{ n \}, N_1 \setminus \{ n \}))$

$M_{\text{ds}} = \bigcup_{n \in N_2} \text{ms}(\{ \pi_2(e) | e \in R \land (\pi_1(e) = n \lor \pi_4(e) = n) \})$

in $(\text{filter}(2, M_{\text{ds}}), N_2)$

Function $\text{collNodes}$ collects all nodes between a given starting one and a set of possible terminal nodes. It aims at identifying the corresponding subgraph. The arguments are a coordination pattern, a starting node, the nodes in path (an empty set at the beginning), the nodes visited and the terminal nodes:

$$\text{collNodes} : \mathcal{P} \times \text{Node} \times \text{Node} \times \text{Node} \times \text{Node} \to \text{Node} \times \text{Node}$$

$$\text{collNodes}(\langle I, O, R \rangle, n, a, v, d) =$$

let $A = \{ x | x \neq \perp \land (n, \ldots, x) \in R \lor (x, \ldots, n) \in R \}$

$(\text{acc}, \text{vis}) = rC(\langle I, O, R \rangle, n, A, v, d)$

in $(a \cup \text{acc}, \text{vis})$

where

$$rC : \mathcal{P} \times \text{Node} \times \text{Node} \times \text{Node} \times \text{Node} \to \text{Node} \times \text{Node}$$

$$rC(\emptyset, n, \emptyset, v, d) = (\emptyset, v)$$

$$rC(\rho, n, (x : xs), v, d) =$$

if $x \in d$ then let $(a, v_1) = rC(\rho, n, xs, v \cup \{ x \}, d)$

in $(\{ n \} \cup a, v \cup v_1)$

else if $x \in v$ then $rC(\rho, n, xs, v \cup \{ x \}, d)$

else let $(r, v_2) = \text{collNodes}(\rho, x, \emptyset, v \cup \{ x \}, d)$

in if $r \neq \emptyset$ then $(r \cup \{ n \}, v_2)$

else $rC(\rho, n, xs, v_2, d)$
Note that these definitions resort to multi-sets, defined, as usual, as functions from the type of interest to the natural numbers (which encode multiplicities). Typical operations on multisets include \(\text{domain}(C) = \{a \in A | C(a) \neq 0\}\) and \(\text{union}, (C_1 \sqcup C_2)(a) = C_1(a) + C_2(a)\). Conversion to sets, and back, are also recorded here as \(ms : 2^A \rightarrow \mathbb{N}^A, ms(A) = |a \rightarrow 1[a \in A]\), and \(filter : \mathbb{N} \times \mathbb{N}^A \rightarrow 2^A\) given by \(filter(i, C) = \{x \in \text{dom}(C) | C(x) \geq i\}\). The latter converts a multi-set into a set, by filtering the elements that occur at least \(i\) times. For a better comprehension of this pattern, the reader may refer to the case study in Section 5.

4 CoopLa: a language for patterns and reconfigurations

Both architectural and reconfiguration patterns can be designed with the help of a domain specific language — CoopLa — and an integrated editor, supplied as a plug-in for Eclipse. It supports syntax colouring and intelligent code-completion and offers during-edition syntax and semantic error checking and error marking for consistent development of patterns. While editing, the tool offers a visualisation of its graph representation, and any change in the code is automatically reflected in this view. Fig. 4 shows a snapshot.

With CoopLa we define communication channels, coordination pattern and reconfigurations.

Channels. Fig. 3 depicts the definition of some of the Reo-like channels introduced above. Note that the lossy channel type extends that of sync (cf., the \texttt{extends} keyword). This means the information flow from \(a\) to \(b\) defined in the latter still applies; only additional behaviour is specified: if there is a request on \(a\) but not on \(b\), data will flow through \(a\) and lost (cf., \texttt{NULL} keyword). Notice the use of \(\texttt{!}\) to explicitly express the absence of requests on \(b\). As another example, consider the drain channel. It has two input ports through which data flows to be lost. The ‘\(|\)’ construct means that both flows are performed in parallel. Finally, the FIFO channel has an internal state of type \texttt{buffer} specified as a sequence of dimension \(N\) and observers \(E\) and \(F\) on which result depends the channel behaviour.

\begin{verbatim}
channel sync(a:b)
  a,b -> flow a to b;
}

channel lossy(a:b) extends sync
  a, !b -> flow a to NULL;
}

channel drain(a,b:)
  a, b -> flow a to NULL | flow b to NULL;
}

channel fifo^N(a:b)
  state: buffer;
  observers: E, F;
    // buffer = ELEM;
    // E = buffer.len = 0;
    // F = buffer.len = N;
    a, !F -> flow a to buffer;
    !E, b -> flow buffer to b;
}
\end{verbatim}

Fig. 3. The sync, lossy, drain and fifo channels in CoopLa.

Coordination patterns. Coordination patterns are defined by composition of primitive channels and patterns previously defined. Declaration of instances is
preceded by the reserved word use. Each instance is declared by indicating (i) the entity name with the ports locally renamed and (ii) a list of aliases (similar to variables in traditional programming languages) to be used in the subsequent parts of the pattern body definition. In case of instantiating a channel with time or structure, it is defined the inherent dimensions, and in some cases, how such structure is initialised (making use of the observers defined for such structure).

Patterns are composed by interconnecting ports declared in their interfaces. This is achieved by the set of primitive reconfigurations introduced in Definition 2. Fig. 4 shows an example of the Sequencer coordination pattern expressed in the context of the tool developed to support CooPLa.

![Fig. 4. Tool Support for CooPLa](image)

Reconfigurations. Reconfigurations in CooPLa are also specified compositionally from the primitives given in Definition 2, or from more complex reconfigurations previously defined. Operators over standard data types (e.g., List, Pair and Triple) can also be used: such is the case, in Fig. 5 of the forall structure which iterates over all elements of a list. Application of a reconfiguration \( r \) to a pattern \( p \) is denoted by \( p \circ r \). Fig. 5 shows an example of two reconfiguration specifications and respective application to instances of coordination patterns. Both Fig. 4 and 5 present parts of the case study addressed next.

5 Example: A fragment of a case-study

This section illustrates the use of architectural and reconfiguration patterns in a typical example of web-service orchestration for system integration. The case-study from where this example was borrowed involved a professional training...
company with facilities in six different locations, which relied on four main software components (all working in complete isolation): an Enterprise Resource Planner (ERP), a Customer Relationship Management (CRM), a Training Server (TS), and a Document Management System (DMS). The expansion of this company entailed the need for better integration of the whole system. This lead to changing components into services and adopting a SOA solution.

Several problems, however, were found during service orchestration analysis. A recurrent one was the lack of parallelism in the business workflow, slowing the whole system down. The user’s information update activity which involves the user update services provided by ERP, CRM and TS components, was one of the tasks affected by such lack of parallel computation, as these services were invoked in sequence.

Let \( \rho \), in Fig. 6, be the coordination pattern (known as a Sequencer) used for sequential service orchestration. Resorting to the reconfiguration patterns introduced in Section 3.2, let us rearrange the coordination policy so that user profiles (in each component) are updated in parallel. A possible solution is obtained by applying the \texttt{implodeP} reconfiguration pattern as \( \rho \circ \texttt{implodeP}(\{j_1, j_3\}, n) \). The following paragraphs show, step-by-step, how to compute the resulting coordination pattern, depicted in Fig. 7. The actual \texttt{CooPLa} script for this reconfiguration is depicted in Fig. 5.

The first argument of \texttt{implodeP} provides the border nodes of the structure one desires to superpose onto the node in the second argument. From Definition 11 we first identify the complete structure to remove, which is composed of the unique names of the channels and their nodes (the border nodes plus some intermediary ones). Operation \texttt{remNodes}, with \( \rho \) and \( \{j_1, j_3\} \) as arguments, starts by identifying the intermediary nodes and channels to retrieve the relevant
Applying the reconfiguration pattern (Fig. 7) which encodes a parallel workflow policy and consequently enforces that no other activity should start before the user’s information is updated. The obvious solution is to delay the flow on port 0, when one of the three operations has been removed from the initial coordination pattern, which is actually the outcome of removing \( f_1 \) from \( \rho \): \[
\rho' = \left\{ \{i, j_2\}, \{\text{crm, erp, ts, o}\}, \left\{ (i, s_1, \text{sync}, j_1), (j_1, s_2, \text{sync, crm}), (j_1, f_1, \text{fifo}, j_2), (j_2, s_3, \text{sync, erp}), (j_2, s_4, \text{fifo}, j_3), (j_3, s_5, \text{sync, ts}) \right\} \right\}
\]

and \( \rho'' \) be the result of removing \( f_2 \) from \( \rho' \), which is actually the outcome of applying the \text{removeP} reconfiguration pattern to \( \rho' \): \[
\rho'' = \left\{ \{i, j_2, j_3\}, \{\text{crm, erp, ts, o}\}, \left\{ (i, s_1, \text{sync}, j_1), (j_1, s_2, \text{sync, crm}), (j_2, s_3, \text{sync, erp}), (j_3, s_4, \text{sync, ts}) \right\} \right\}
\]

After removing the channels, set \( N_2 \) is updated, to delete nodes that have been removed from the initial coordination pattern, \( \rho \). In this case, \( N_2 \) remains unchanged, then:

\[
M = \{j_1, j_2, j_3\} \cap \{i, j_1, \text{crm, j2, erp, j3, ts, j3, o}\} = \{j_1, j_2, j_3\}
\]

Finally, we merge the nodes of \( M \) with node \( n \), and obtain the desired coordination pattern (Fig. 7) which encodes a parallel workflow policy and consequently allows for the update of user’s information in parallel.

The resulting pattern actually does the job: the three user update services are called simultaneously, and the flow continues to the output port \( o \), which enables contiguous activities. However, it does not cope with another requirement enforcing that no other activity should start before the user’s information is updated. The obvious solution is to delay the flow on port \( o \), until the three services
provide a finish acknowledgement. A new reconfiguration is, therefore, necessary: we proceed by overlapping a Synchroniser pattern \( \rho_s \) (see Fig. 8). The CooPLa specification of this reconfiguration is shown in Fig. 5. The idea is to connect nodes \( o \) and \( a \) in such a way that all other input ports of \( \rho_s \) are free to connect to the feedback service interface of the CRM, ERP and TS components. Fig. 9 depicts the result of performing \( \rho_1 \bullet \text{overlap}(\rho_s, \{o, a\}) \). From Definition 8, we start by computing \( \rho_1 \bullet \text{par}(\rho_s) \), which yields the following pattern:

\[
\rho'' = \left\{ \begin{array}{c}
  (i, s_1, \text{sync}, n),
  (n, s_2, \text{sync}, \text{crm}),
  (n, s_3, \text{sync}, \text{erp}),
  (n, s_4, \text{sync}, \text{ts}),
  (n, s_5, \text{sync}, \text{crm}),
  (n, s_6, \text{sync}, \text{erp}),
  (n, s_7, \text{sync}, \text{ts}),
  (n, s_8, \text{sync}, \text{crm}),
  (n, s_9, \text{sync}, \text{erp}),
  (n, s_10, \text{sync}, \text{ts}),
  (a, s_9, \text{fifo}, k_1),
  (a, s_{10}, \text{fifo}, k_2),
  (k_1, d_3, \text{drain}, \perp),
  (k_2, d_3, \text{drain}, \perp),
  (k_3, d_2, \text{drain}, \perp),
  (k_3, d_1, \text{drain}, \perp),
  (k_4, d_1, \text{drain}, \perp),
  (\text{crm}_1, s_8, \text{sync}, k_4)
\end{array} \right\}
\]

Finally, we merge nodes \( o \) and \( a \) together into a node \( j \), by performing \( \rho'' \bullet \text{join}(\{o, a\}, j) \). The result is presented in Fig. 9, which is actually the coordination pattern meeting the requirement of not allowing other activities to start before the user's information update in the CRM, ERP and TS components is completed.

Fig. 7. After imploding the Sequencer: the Parallel Split coordination pattern

Fig. 8. The Synchroniser Coordination Pattern
Reconfiguration mechanisms for service coordination

6 Conclusions

6.1 Related work

Reconfigurations in SOA are, most of the times, focused on replacing services, or modifying their connections to the coordination layer. Often they neglect structural changes in the actual interaction layer itself [8,12]. In [14,13], however, the authors highlight the role played by software connectors during runtime architectural changes. Although these changes are again focused on the manipulation of components, they recognise that connectors are also amenable to contextual adaptations in order to keep the consistency of the architecture.

Reference [19] resorts to category theory to model software architectures as labelled graphs of components and connectors. Reconfigurations are modelled via algebraic graph rewriting rules. This approach has some points of contact with our strategy. In [11,10], a similar approach is adopted, but in the context of Reo. The authors rely on high-level replacement systems, more precisely on typed hypergraphs, to describe Reo connectors (and architectures, in general). In this perspective, vertices are the nodes and (typed hyper-) edges are communication channels and components. Reconfiguration rules are specified as graph productions for pattern matching. This approach performs atomic complex reconfigurations, rather than a sequence of basic modifications, which is stated as an advantage for maintaining system consistency. Nevertheless, the model may become too complex even when a simple primitive operation needs to be applied.

Differently, in [4] architectures are modelled as Reo connectors, and no information on components is stored in the model. The model is a triple composed of channels with a type and distinct named ports, a set of visible nodes and a
set of hidden nodes. Their model is similar to ours, but for the distinction introduced here between input and output nodes and the need we avoid to be explicit on the hidden nodes of a pattern. Although a number of primitive transformations are proposed, this work, as most of the others, do not consider ‘big-step’ reconfigurations which seems a severe limitation in practice.

6.2 Summary and future work

The paper introduces a model for reconfiguration of coordination patterns, described as a graphs of primitive channels. It is shown how typical reconfiguration patterns can be expressed in the model by composition of elementary transformations. CoopLa provides a setting to animate and experiment reconfigurations upon typical coordination patterns. We are currently involved in their classification in a suitable ontology. What is still missing, however, is the inclusion in the model of automatic assessing mechanisms to assess reconfigurations semantically and trigger their application. This concern is orthogonal to the work presented in this paper.

References

13. Nenad Medvidovic. ADLs and dynamic architecture changes. In *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints ’96) on SIGSOFT ’96 workshops*, ISAW ’96, pages 24–27, New York, NY, USA, 1996. ACM.
Linking the Semantics of BPEL using Maude

Peng Liu and Huibiao Zhu

Shanghai Key Laboratory of Trustworthy Computing
Software Engineering Institute, East China Normal University
3663 Zhongshan Road (North), Shanghai, China, 200062
{liup, hbzhu}@sei.ecnu.edu.cn

Abstract. Web services have become more and more important in these years. It is of key importance for enterprise web applications to combine different services available to accomplish complex business process. BPEL4WS (BPEL) is the OASIS standard for web services composition and orchestration. It contains several distinct features, including scope-based compensation and fault handling mechanism. We have already studied the semantics for BPEL, including the operational semantics, algebraic semantics and their linking theory.

This paper considers the mechanical approach to linking the operational semantics and algebraic semantics for BPEL. Our approach is to generate operational semantics from algebraic semantics, and to use equational and rewriting logic system Maude to mechanize the linking between the two semantics. Firstly, we investigate the algebraic laws in the Maude approach. Based on the algebraic semantics, the generation of head normal form is explored. Secondly, we consider the Maude approach to deriving the operational semantics from algebraic semantics, where the derivation strategy is based on the concept of head normal form. Our mechanical approach using Maude can visually show the head normal form of each program, as well as the execution steps of a program based on the derivation strategy. Finally, we also mechanize the derived operational semantics. The results mechanized from the second and third exploration indicate that the transition system of the derived operational semantics is the same as the one based on the derivation strategy.

1 Introduction

Web services and other web-based applications have been becoming more and more important in practice. In this flowering field, various web-based business process languages have been introduced, such as XLANG [30], WSFL [23], BPEL4WS (BPEL) [11] and StAC [4], which are designed for the description of services composed of a set of processes across the Internet. Their goal is to achieve the universal interoperability between applications by using web standards, as well as to specify the technical infrastructure for carrying out business transactions. BPEL4WS (BPEL) is the OASIS standard for web services composition and orchestration. It contains several distinct features, including scope-based compensation and fault handling mechanism.

The most important feature of BPEL is that it supports the long-running
interactions involving two or more parties. Therefore, it provides the ability to define fault and compensation handing in application-specific manner, resulting in a feature called Long-Running (Business) Transactions (LRTs). The concept of compensation is due to the use of Sagas [3, 12] and open nested transactions [26]. The fault analysis has been considered in [27, 29] for compensation processes. In addition, BPEL provides two kinds of synchronization techniques for parallel processes. In our model, shared-labels are introduced for the synchronization between a process and its partners within a single service, while channel communications are introduced for message transmission between different services.

We have already studied the various semantics for BPEL [28, 34, 14, 35]. The operational semantics was studied using SOS, and bisimulation is explored for BPEL programs, which is considered in a two-layer form, one is for the equivalence for the compensation lists, the other is for BPEL programs. The denotational semantics was studied using the UTP [19] approach. A set of algebraic laws was explored, and the their correctness is based on the two approaches, operational semantics and denotational semantics respectively. The three semantics should provide the same understanding of the language from different viewpoints and they should be consistent. Therefore, the relating of these three semantics is a challenging task. We have also theoretically investigated some of the linking theories between the three semantics of BPEL [32, 33].

This paper studies the mechanical approach to linking theory of the operational and algebraic semantics for BPEL. Our approach is to generate operational semantics from algebraic semantics. We apply the mechanical method to support the semantic linking by using the equational and rewriting logic system Maude [9, 10]. Firstly, we investigate the algebraic semantics for BPEL. In order to generate the head normal form of each program, four typical forms are introduced. The concept of head normal form supports the linking between algebraic and operational semantics, and its generation is implemented in Maude. Secondly, we investigate the derivation of operational semantics from algebraic semantics. The derivation strategy is defined based on the head normal form and it is encoded in Maude. Finally, based on the derivation strategy, a transition system can be derived (i.e., derived operational semantics). We animate the generated operational semantics in Maude. The results mechanized from the second and third exploration indicate that the transition system of the derived operational semantics is the same as the one based on the derivation strategy.

The remainder of this paper is organized as follows. Section 2 introduces the language BPEL, and we implement its syntax in Maude. Section 3 explores the algebraic semantics for BPEL implemented in Maude. In particular, we study the laws for parallel and scope structure of BPEL. They are the key features of BPEL and deserve a detailed study. Meanwhile, in this section we also study the generation of head normal form mechanically. In section 4 we provide the derivation strategy for deriving operational semantics from algebraic semantics. Our approach is based on the head normal form of each program. An operational semantics can be derived. For the derived operational semantics, we study its
animation in section 5. Section 6 discusses the related work about web services and semantics linking. Section 7 concludes our paper and discuss some future work.

2 The Syntax of BPEL and its Implementation in Maude

2.1 Introduction of BPEL

We have proposed a BPEL-like language, which contains several interesting features, such as scope-based compensation and a fault handler mechanism. The categories of syntactic elements of the language are as follows:

\[
BA ::= \text{skip} \mid x := e \mid \text{rec } a \ x \mid \text{rep } a \ x \mid \text{throw}
\]

\[
A ::= BA \mid g \circ A \mid b \triangleright l \mid A \mid A < b \triangleright A \mid b \ast A \mid A \parallel A \\
| A \sqcap A \mid \text{undo} \mid \{A? A, A\}
\]

where:

- \(x := e\) assigns the value of \(e\) to local variable \(x\). \(\text{skip}\) behaves the same as \(x := x\). Activity \(\text{throw}\) indicates that the program encounters a fault immediately. In order to implement the communications between different services, two statements are introduced; i.e., \(\text{rec } a \ x\) and \(\text{rep } a \ x\). Command \(\text{rec } a \ x\) represents the receiving of a value through channel \(a\), whereas \(\text{rep } a \ x\) represents the output of value of \(x\) via channel \(a\).

- Several constructs are similar to those in traditional programming languages. \(A; B\) stands for sequential composition. \(A < b \triangleright B\) is the conditional construct and \(b \ast A\) is the iteration construct. \(A \sqcap B\) stands for the nondeterministic choice.

- Shared-labels are introduced to implement the synchronization between a process and its partner. \(g \circ A\) awaits the Boolean guard \(g\) to be set true, where \(g\) is composed of a set of source links; i.e., a set of read only shared-labels. \(b \triangleright l\) assigns the value of \(b\) to label \(l\).

- \(\{A? C, F\}\) stands for the scope-based compensation statement, where \(A\), \(C\) and \(F\) stand for the primary activity, compensation program and fault handler correspondingly. If \(A\) terminates successfully, program \(C\) is installed in the compensation list for later compensating. On the other hand, if \(A\) encounters a fault during its execution, the fault handler \(F\) will be activated. Further, the compensation part \(C\) does not contain scope activity. On the other hand, statement \(\text{"undo"}\) activates the execution of the programs in the compensation list in reverse order of their installed sequence.

- \(A \parallel B\) stands for the parallel composition. Its local variable set is the union of the corresponding sets for the two components. For a shared label, it can be owned by one parallel component; i.e., it can only be written by one parallel component.

For exploring the parallel expansion laws, our language is enriched with the concept of guarded choice, expressed in the form:

\[
\{h_1 \rightarrow P_1\} \ldots \ldots \{h_n \rightarrow P_n\}
\]
where:

1. $h_i$ can be a skip guard, expressed as $b_i \& \text{skip}$, where $b_i$ is a Boolean expression and satisfies the condition “$\forall i, b_i = \text{true}$”.

2. $h_i$ can also be a communication guard, expressed as $\text{rec } a x$ and $\text{rep } a x$.

3. $h_i$ can also be a Boolean guard $@((g_i))$; i.e., waiting for $g_i$ to be fired.

2.2 Rewriting Logic and the Maude System

Rewriting logic is a general semantic and logic framework that supports defining semantics of programming languages and models of concurrency. Membership equational logic is included in rewriting logic. Thus rewriting logic supports both static and dynamic specifications. A rewrite theory in rewriting logic is a 4-tuple $R = (\Sigma, E \cup A, \emptyset, R)$, where $(\Sigma, E \cup A)$ is a membership equational logic.

Members are defined in the static part of a theory. $\Sigma$, which defines the sorts, subsorts, and operators in the theory, is the fundamental part called the signature. $E$ is the collection of equations ($t = t'$) and membership ($t : s$) $^1$ (possibly conditional) axioms of the theory. It defines simplification rules regarding the signature. $A$ is a set of equational attributes declared for operators, including $\text{assoc}$ (associative), $\text{comm}$ (commutative), $\text{id}$ ($t$ as a identity of the operator) and so on. Particularly, the attribute $\text{ctor}$ (short for constructor) declares the fundamental elements of a sort.

The third component $\emptyset$ of the rewriting theory is the function defining frozen arguments of operators. $R$ is a collection of (possibly conditional) rules specifying the dynamic part of the theory. A rewriting rule $t \rightarrow t'$ represents a transition of a concurrent system, and can naturally depict transitions of operational semantics, which we will describe later in this paper.

Maude is a high-performance implementation of rewriting logic as well as the underlying MEL sublogic. Its syntax is so simple that it is almost identical with mathematical notations. And with rewriting logic as its fundamental logic, it is very expressive, specifying semantics of programming languages and modeling concurrent systems.

Maude supports two levels of declaration, functional modules and system modules. A functional module is declared by the syntax $\text{fmod Module-Name is} \ldots \text{endfm}$, where the dots denote the declaration of sorts, subsorts, operations, equations and memberships. Functional modules are implementation of MEL theory. System modules are declared by $\text{mod is Module-Name} \ldots \text{endm}$, where rules can be declared in addition to the parts of functional modules. System modules are implementation of rewriting logic.

2.3 Implementation of Syntax of BPEL in Maude

Utilizing the sort oriented system provided by Maude, it is rather convenient to implement the syntax of BPEL. First of all, we use the sort $\text{Assignment}$ to

\footnote{where $t$ and $t'$ as terms and $s$ as a sort}
define the \( x := e \) and \( \text{skip} \). In order to handle the update of source links, we define a unique sort \( \text{UpdateOfLabel} \) to distinguish it from the assignment of local variables. Communications on channels are defined by the sort \( \text{Channel} \), including the two basic communication primitives, i.e., \( \text{rep} \) and \( \text{rec} \). \( \text{throw} \) and \( \text{undo} \) are defined by the sorts \( \text{ThrowCommand} \) and \( \text{UndoCommand} \) respectively. The syntax elements mentioned above constitute the sort \( \text{PrimitiveCommand} \), by declaring their corresponding sorts as subsorts of \( \text{PrimitiveCommand} \).

sorts Assignment UpdateOfLabel Channel ThrowCommand UndoCommand .
sort PrimitiveCommand .
subsort Assignment AssignmentOfLabel Channel < PrimitiveCommand .
subsort ThrowCommand UndoCommand < PrimitiveCommand .
op \_ := \_ : Qid Exp -> Assignment [ctor prec 1] .
op \_ /\_ : BoolExp Qid -> UpdateOfLabel [ctor prec 1] .
op rec \_ \_ : Qid Qid -> Channel [ctor prec 1] .
op rep \_ \_ : Qid Qid -> Channel [ctor prec 1] .
op throw : -> ThrowCommand .
op undo : -> UndoCommand .

The structured construction of BPEL programs are defined by the sort \( \text{Program} \). To be mentioned, three particular states of programs are defined to be basic elements of \( \text{Program} \). They are \( \text{nil} \), \( \text{special} \) and \( \text{fault} \), representing that the program is completed, in the compensating status and confronted a fatal fault. The last two are the essential parts of the scope-based compensation and fault handler mechanism.

\( g \circ A \) is represented as \( \varnothing(g) \ A \) in our code, and \( \varnothing(g) \) is of sort \( \text{LabelGuard} \). And other syntax constructions are defined as in most programming languages.

fmod PROGRAM is ...

subsort PrimitiveCommand < Program .
op nil special fault : -> Program [ctor] .
op \_ \_ : LabelGuard Program -> Program [ctor prec 5] .
op \_ /\_ : Program Program -> Program [ctor prec 10] .

---- Nondeterministic Choice
op \_ ; \_ : Program Program -> Program [ctor gather (e E) prec 40] .
op if \_ then \_ else \_ : BoolExp Program Program -> Program [ctor prec 20] .
op while \_ do \_ : BoolExp Program Program -> Program [ctor prec 20] .
op \_ || \_ : Program Program -> Program [ctor prec 30] .
op \_ \_ \? \_ \_ : Program Program Program -> Program [ctor prec 20] .
endfm

3 Algebraic Semantics and Head Normal Form for BPEL

In this section we study the algebraic semantics and head normal forms for BPEL. The concept of head normal form is used to build the link between the operational semantics and algebraic semantics. Firstly, we introduce four
3.1 Typical Forms

In order to explore the head normal form, we introduce four typical forms. We define a sort NormalForm declared to represent these four typical forms. Below describes how the four typical forms are represented as elements of NormalForm.

The first form stands for the general guarded choice, as mentioned above. For each guarded component, the corresponding guard can be a skip guard, event guard or communication guard.

\[(form-1) \quad \left\{ \begin{array}{l}
\left[ i \in I \{ b_i \text{ and skip } \rightarrow P_i \} \right] \\
\left[ j \in J \{ @\left( g_j \right) \rightarrow Q_j \} \right] \\
\left[ l \in L \{ \text{comm } a_i x_l \rightarrow R_l \} \right]
\end{array} \right. \]

where, comm can be rec or rep

\[\text{sorts LabelGuard Guard .} \]

\[\text{subsort Channel LabelGuard < Guard .} \]

\[\text{op } \_\&\text{skip : BoolExp } \rightarrow \text{Guard } [\text{ctor}]. \]

\[\text{op } @(_) : \text{ParterLink } \rightarrow \text{LabelGuard } [\text{ctor}]. \]

\[\text{subsort GuardedChoice < NormalForm .} \]

\[\text{op } _\Rightarrow _ : \text{Guard Program } \rightarrow \text{GuardedChoice } [\text{ctor}]. \]

\[\text{op } &: \rightarrow \text{GuardedChoice } [\text{ctor}]. \]

\[\text{op } _[] : \text{GuardedChoice GuardedChoice } \rightarrow \text{GuardedChoice[ctor assoc comm id:] $} \]

\[(form-2) \quad X \rightarrow P \]

where \(X\) can be of the form \(\text{assign}(x, e)\), \(\text{updatelabel}(l, b)\) or \(\text{compen}(C)\), which we introduce to represent actions of BPEL programs except for the communication actions. \(\text{assign}(x, e)\) stands for assigning value \(e\) to local variable \(x\). \(\text{updatelabel}(l, b)\) means the update of label \(l\) with boolean value \(b\). We use \(\text{compen}(C)\) to stand for recording program \(C\) at the end of the compensation list

\[\text{op } \text{assign}(_, _) : \text{Qid Exp } \rightarrow \text{Head } [\text{ctor}]. \]

\[\text{op } \text{updatelabel}(_, _) : \text{Qid BoolExp } \rightarrow \text{Head } [\text{ctor}]. \]

\[\text{op } \text{compen}(_) : \text{Program } \rightarrow \text{Head } [\text{ctor}]. \]

\[\text{op } _\Rightarrow _ : \text{Head Program } \rightarrow \text{NormalForm } [\text{ctor}]. \]

\[(form-3) \quad \text{throw} \]

\[\text{subsort ThrowCommand < NormalForm .} \]

\[(form-4) \quad \text{undo} \]

\[\text{subsort UndoCommand < NormalForm .} \]

Now we start to consider the algebraic laws. First we consider the algebraic laws for sequential composition. It is distributive backward through guarded choice and the second typical form. Further, \(\text{throw}\) and \(\text{undo}\) are both a left zero of sequential composition.

\[\text{eq } \{ h \Rightarrow P \} ; Q = \{ h \Rightarrow (P ; Q) \} . \]
eq \{(h \Rightarrow P) \sqcap GC\} ; Q = (\{h \Rightarrow P\} ; Q) \sqcap (GC ; Q) . ^2

eq (X \Rightarrow P) ; Q = X \Rightarrow (P ; Q) .

eq \text{throw} ; P = \text{throw} .

eq \text{undo} ; P = \text{undo} .

As for function \textit{seq}, it defines that \textit{nil} and \textit{throw} are unit and zero of sequential composition respectively.

eq \text{nil} ; Q = Q .
eq \text{throw} ; Q = \text{throw} .

In the case of conditional branch and iteration, there are algebraic laws defined as following:

\begin{align*}
\text{eq } & \text{if } b \text{ then } P \text{ else } Q = \{b \& \text{skip} \Rightarrow P\} \sqcap \{(\neg b \& \text{skip} \Rightarrow Q)\} . \\
\text{eq } & \text{while } b \text{ do } P = \{b \& \text{skip} \Rightarrow (P \sqcup \text{while } b \text{ do } P)\} \sqcap \{(\neg b \& \text{skip} \Rightarrow \text{nil})\} .
\end{align*}

Scope construct and parallel composition are two main features of BPEL, so we will introduce them in detail in the following two subsections.

### 3.2 Scope

Scope-based compensation and fault handling mechanism are the core features of BPEL, so we describe them here specially. First we introduce the action called \textit{compen}(C), installing the program \textit{C} into the compensation list.

\begin{align*}
\text{(scope-1) } \{A ? C, F\} = \text{df } (A ; \text{compen}(C)) \circ F
\end{align*}

When program \textit{A} terminates successfully, \textit{compen}(C) will be activated. In this sense, it can be interpreted as a sequential composition. In the meantime, if \textit{A} encounters a fault, fault handler \textit{F} will take charge to do protection actions.

For the operator \textit{\circ}, it satisfies the following algebraic laws.

\begin{align*}
\text{eq } & \{h \Rightarrow P\} \circ Q = \{h \Rightarrow (P \circ Q)\} . \\
\text{eq } & \{(h \Rightarrow P) \sqcap GC\} \circ Q = (\{h \Rightarrow P\} \circ Q) \sqcap (GC \circ Q) . \\
\text{eq } & (X \Rightarrow P) \circ Q = X \Rightarrow (P \circ Q) . \\
\text{eq } & \text{throw} \circ P = P . \\
\text{eq } & \text{undo} \circ P = \text{undo} .
\end{align*}

As for function \textit{fau}, it defines that \textit{nil} and \textit{throw} are zero and unit of operator \textit{\circ} respectively, just as the opposite of sequential composition.

\begin{align*}
\text{eq } & \text{nil} \circ Q = \text{nil} . \\
\text{eq } & \text{throw} \circ Q = Q .
\end{align*}

### 3.3 Parallel Composition

In this subsection, we will explain how the normal forms of two initially deterministic processes are composed together. We consider the four typical forms in

\begin{align*}
G1 \ G2 \ GC \ \text{GC1} \ \text{GC2} \text{ are all variables of sort GuardedChoice in this paper.}
\end{align*}
turn to show the parallel composition. First, we consider the algebraic laws for
two parallel programs. nil is a unit of parallel composition.

\[ \text{eq } \text{nil } || \text{ Q } = \text{ Q } . \]

Then we introduce a fundamental function \( \text{par} \) which defines how to combine a
guarded choice with a program. It has to distribute the program to every com-
ponent of the guarded choice.

\[ \text{eq } (\{h => P\} \[ G1 \] || Q = \{h => P || Q\} \[ G1 || Q \} . \]

\[ \text{eq } $ || \text{ Q } = $ . \]

The parallel expansion of normal forms are interpreted by the operator /// fol-
lowing.

\[ \text{op } _{}/_{}/_{}/_{/} : \text{NormalForm NormalForm NormalForm NormalForm -> NormalForm} . \]

At first, we consider that the two parallel components are both in the first typi-
cal form, i.e., guarded choice. It is implemented by the function \( \text{foo} \). In order to
remember the initial form of the two parallel components, we pass an additional
copy of them to the function \( \text{foo} \).

\[ \text{eq } G1 // G2 = \text{foo}(G1,G2,G1,G2) . \]

\[ \text{op } \text{foo}(_{/}_{/}_{/}_{/}) : \text{GuardedChoice GuardedChoice GuardedChoice GuardedChoice -> GuardedChoice} . \]

(para-1) If two processes running in parallel have a common channel and are to
communicate through the channel, the message passing can be interpreted as an
assignment.

\[ \text{eq } \text{foo}(G1 \[ \{\text{rep a x => R}\}, G2 \[ \{\text{rec a y => T}\}, GC1, GC2) = } \]

\[ \text{foo}(G1,G2,GC1,GC2) \[ \{\text{t &skip => (y := x ; (R || T)}\} . \]

\[ \text{eq } \text{foo}(G1 \[ \{\text{rec a x => R}\}, G2 \[ \{\text{rep a y => T}\}, GC1, GC2) = } \]

\[ \text{foo}(G1,G2,GC1,GC2) \[ \{\text{t &skip => (x := y ; (R || T)}\} . \]

(para-2) If both do not have any communication through the same channel right
now, they can be composed together by installing one program into another’s
guarded choice. Here, we will use the initial forms of two components which is
passed as additional parameters.

\[ \text{eq } \text{foo}(G1,G2,GC1,GC2) = (G1 || GC2) \[ (G2 || GC1) \[ \text{wise} \} . \]

(para-3) Next, we consider the case that one belongs to the first kind of typical
form and the other one is in the second kind. This situation is of the normal
parallel expansion law. Both can have a chance to be scheduled.

\[ \text{eq } GC // (X => Q) = (GC || (X ; Q)) \[ \{\text{t &skip => (X ; GC || Q)}\} . \]

(para-4) If one is \text{throw}, the whole is also \text{throw}.

\[ \text{eq } \text{N } // \text{ throw } = \text{ throw} . \]

(para-5) If both parallel components are of the second typical form, they can be
expanded rather easily.

\[ \text{eq } (X => P) // (Y => Q) = \{\text{t &skip => (X ; (P || Y;Q))}\}
\[ \\[ \{\text{t &skip => (Y ; (X;P || Q))}\} . \]
3.4 Introduction of Summation

The four typical forms can describe most BPEL programs except for the ones containing nondeterministic choices. In order to include nondeterministic choices, we introduce a summation of normal forms. The basic idea is to distribute current possible choices to the front, summing up by the summation operator.

```
sort Summation.
subsort Program < Summation.

op _ (*)_ : Summation Summation -> Summation [assoc comm].
```

We define it as associative and commutative, to reflect the essence of nondeterministic choice. The laws of summation are defined as:

```
eq (P (*) Q) ; R = P ; R (*) Q ; R.
eq (P (*) S1) ; R = (P ; R) (*) (S1 ; R) [owise].
eq (P (*) Q) ° R = (P ° R) (*) (Q ° R).
eq (P (*) S1) ° R = (P ° R) (*) (S1 ° R) [owise].
```

3.5 Head Normal Form

With NormalForm and Summation, we are able to translate all BPEL programs to the two kinds of forms. And this is the essence of the head normal form and the algebraic semantics. We combine them together as sort HeadNormalForm. Operator HF is declared to compute head normal forms of programs.

```
sort HeadNormalForm.
subsort NormalForm Summation < HeadNormalForm.

op HF(_) : Program -> HeadNormalForm.
```

Below we define how head normal forms of each BPEL program can be generated based on the syntax structure.

Assignment $x := e$ is transformed to a single guarded choice with $true$ & $skip$ as its guard, and $assign(x,e)$ as its subsequent behavior.

```
eq HF(x := e) = {t & skip => assign(x,e)}.
```

Communication action on channels are expressed as a guarded choice with communication guard and $nil$ as its subsequent behavior.

```
eq HF(rec a x) = {rec a x => nil}.
eq HF(rep a x) = {rep a x => nil}.
```

$throw$ and $undo$ are just as they are, and the update of label $b /\!\!>$ $l$ is similar to assignment. The transformation of $g \circ A$ is simply to put the guard $g$ in the front and $A$ to the back as a guarded choice component.

```
eq HF(throw) = throw.
eq HF(undo) = undo.
eq HF(b /\!\!> l) = {t & skip => assignlabel(l,b)}.
eq HF(\$g(p)) P = {\$g => p}.
```

As for sequential composition $P; Q$, if the head normal form of $P$ is a guarded
choice or \( X \rightarrow P \), then we can just append \( Q \) to the head normal form of \( P \). On the other hand, if the head normal form of \( P \) is throw or undo, then the head normal form of \( P; Q \) is also throw or undo respectively. If the head normal form of \( P \) is in the form of summation, the transform is committed on each part of the summation.

\[
eq HF(P ; Q) = HF(P) ; Q.
\]

For conditional branch and iteration, their head normal form is the guarded choice based on the Boolean condition.

\[
eq HF(if b then P else Q) = \{ b &skip => P \} \sqcup \{ !b &skip => Q \}.
\]

For nondeterministic choice \( P \sqcap Q \), we have to take whether \( P \) or \( Q \) is already a nondeterministic choice into consideration. If the head normal forms of \( P \) and \( Q \) are both summations of a set of processes, then the head normal form of \( P \sqcap Q \) is the sum of their head normal forms. If the head normal form of one of them is a single deterministic process, for example \( Q \), then the result is the sum of \( HF(P) \) and \( Q \). Otherwise, the result is the sum of \( P \) and \( Q \).

\[
ueq HF(P \leftrightarrow Q) = HF(P) (*) HF(Q) \quad \text{if numofsumm}(HF(P)) > 1 \quad \land \quad \text{numofsumm}(HF(Q)) > 1.
\]

\[
ueq HF(P \leftrightarrow Q) = HF(P) (*) Q \quad \text{if numofsumm}(HF(P)) > 1 \quad \land \quad \text{numofsumm}(HF(Q)) = 1.
\]

\[
eq HF(P \leftrightarrow Q) = P (*) Q \quad \text{[owise]}
\]

***numofsumm: the number of initial deterministic processes composing the head normal form

***if numofsumm(HF(P)) > 1, then HF(P) is already a summation of a set of processes

Parallel is closely related to nondeterministic choice, in the sense that if one of the two processes concurrently executing is nondeterministic, we will put the nondeterministic choice at the front, which is implemented by the function cross. If both are deterministic, we apply the parallel expansion law, expressed by the expansion operator // described in subsection 3.3.

\[
ueq HF(P || Q) = cross(HF(P),HF(Q)) \quad \text{if numofsumm}(HF(P)) > 1 \quad \land \quad \text{numofsumm}(HF(Q)) > 1.
\]

\[
eq HF(P || Q) = cross(HF(P),Q) \quad \text{if numofsumm}(HF(P)) > 1.
\]

\[
eq HF(P || Q) = HF(P) // HF(Q) \quad \text{[owise]}
\]

\[
\text{op cross}(_ , _) : \text{Summation Summation} \rightarrow \text{Summation}.
\]

\[
eq cross(P , Q) = par(P,Q).
\]

\[
eq cross(P , Q(*)S1) = par(P,Q) (*) cross(P,S1).
\]

\[
eq cross(P(*)S1 , S2) = cross(P,S1) (*) cross(S1,S2).
\]

The head normal form of scope \{ \( A ? C, F \) \} can be expressed as the introduced form in the subsection Scope.

\[
eq HF(\{A ? C,F\}) = HF(A ; \text{compen}(C)) * F).
\]

For assign\((x,e)\), updatelabel\((l,b)\) and compen\((C)\), their head normal forms are defined as:
eq HF(X) = X => nil.

3.6 Example

Here, we give an example to show the head normal forms of BPEL programs implemented in Maude. Since parallel and scope structure are the key features, we concentrate on them and explore the results.

First, take this parallel program as an example,

\( (\text{rec } a \ x ; \text{if } x < 0 \ \text{then } x := -x \ \text{else } x := x + 1 ; \text{rep } a \ x) \ || \ (\text{rep } a \ y ; \text{rec } a \ y) \)\)

Two processes are running in parallel, and will do a receive/reply communication. The first process waits for a reply action on channel \( a \) and once it receives a value, it will do the corresponding operations and reply the new value through channel \( a \). In the mean time, the second process will reply a value through channel \( a \) and waits for a reply. The head normal form of this program showed in Maude is as:

result GuardedChoice:
\{ t &skip => 'x:='y ; (if 'x<0 then 'x:=(-'x) else 'x:=('x+1) ; rep'a'x) || rec'a'y) \}

As we can see from the head normal form, the receive/reply communication is interpreted as an assignment of local variable.

Now we consider the scope structure.

\{ rec a x ; if x > 10 then throw else x := x + 1 ? x := x - 1, x := 0 \}

In this scope structure, the process first receives a value. If it is larger than 10 it will throw a fault otherwise it will increase it by one. The compensating program is to decrease the value, and the fault handler is to assign the value to its initial value zero. The head normal form is:

result GuardedChoice:
\{ rec'a'x => ((if 'x>10 then throw else 'x:=('x+1)) ; compen('x:=(x-1)) * 'x:=0) \}

The initial action is \( rec \ a \ x \) which is normal, so the rest of the program under the scope is still protected by the fault handler \( x := 0 \), which we can see from the operator \(*\).

4 Generating Operational Semantics from Algebraic Semantics

4.1 Configuration and Transition Types

With the rewriting rules in Maude, it is natural to implement operational semantics. First, we introduce four transition types of BPEL. They are described using the well-known Structural Operational Semantics (SOS).

\[ C \xrightarrow{tau} C' \quad \text{or} \quad C \xrightarrow{c} C' \quad \text{or} \quad C \xrightarrow{v} C' \quad \text{or} \quad C \xrightarrow{\text{a.m}} C' \]
The first transition with $\tau$ action represents the nondeterministic selection. The second with $c$ action represents a transition for local variable assignment while the third with $v$ action represents the update of a shared label. The last is used for communication actions, where $a.m$ is an action $rep$ or $rec$. The action is declared by sort $Act$.

\[
\begin{align*}
\text{op } \tau & : \rightarrow Act \ [\text{ctor}] . \\
\text{op } c & : Qid \ Qid \ Int \rightarrow Act \ [\text{ctor}] . \ ---- \ rep : \text{Variable Channel Value} \\
\text{op } v & : Qid \ Qid \ Int \rightarrow Act \ [\text{ctor}] . \ ---- \ rec : \text{Variable Channel Value}
\end{align*}
\]

The configuration is composed of four parts, the program, state of local variables, state of shared labels, and compensation list, represented as $\langle P, \sigma, L, Cpens \rangle$.

\[
\begin{align*}
\text{op } < \_,\_,\_,\_ > & : \text{Program Env Labels Cpenslist} \rightarrow \text{Config} \ [\text{ctor}] . \\
\text{op } \{\_\} & : \text{Act Config} \rightarrow \text{Config} \ [\text{ctor frozen}] .
\end{align*}
\]

In order to show what kind of transition it is in Maude, we declare two kinds of configurations as shown above. Rewriting rules are a natural way to represent steps of transitions. We use the configuration with an act in the front as the right side of a transition. In this way, we can capture explicitly the type of a transition.

### 4.2 Derivation Strategy

We have described in detail how to generate the head normal form of a BPEL program in section 3. In this subsection we consider how to derive operational semantics from the head normal form, which we call it as derivation strategy.

First, if the head normal form of a program is a summation of at least two programs, it will do a nondeterministic selection among the programs composing the summation and reach any one of them. It takes a $\tau$ action as described by rule $[0]$.

If the head normal form is in the form of guarded choice, it can do four kinds of transitions according to the guarded components in the guarded choice. The transitions are as $[a1]$-$[a4]$.

For the case of the other two typical forms, the update of local variables and shared labels can be done immediately. $\text{compen}(C)$ is to install the compensation program $C$. For a program whose head normal form is $\text{throw}$, it has encountered fatal error and has to be terminated. If the head normal form is $\text{undo}$, the program will do compensation programs installed in the compensating list in the reverse order, which is the core semantics in the compensating mechanism.

\[
\begin{align*}
\text{crl } [0] & : \langle P,\sigma,\text{label},\text{cpens} \rangle \Rightarrow \{\tau\}\langle Pi,\sigma,\text{label},\text{cpens} \rangle \\
\text{crl } [a1] & : \langle P,\sigma,\text{label},\text{cpens} \rangle \Rightarrow \{c\}\langle Pi,\sigma,\text{label},\text{cpens} \rangle \\
& \quad \text{if } \{b \& \text{skip} \Rightarrow Pi\} \ [] \ G := \text{HF(P) } \& \ b[\sigma] \ . \\
\text{crl } [a2] & : \langle P,\sigma,\text{label},\text{cpens} \rangle \Rightarrow \{c\}\langle R,\sigma,\text{label},\text{cpens} \rangle \\
& \quad \text{if } \{g(g) \Rightarrow R\} \ [] \ G := \text{HF(P) } \& \ g[\text{label}] \ . \\
\text{crl } [a3] & : \langle P,\sigma,\text{label},\text{cpens} \rangle
\end{align*}
\]
\[ x \rightarrow x \land a \rightarrow (\sigma).x \]\n\[ \text{if}\ \{\text{rep a x} \rightarrow R\} \\text{G} := \text{HF}(P) .\]

crl [a4] : \< P,\sigma,label,cpens >
\[ \rightarrow x \rightarrow a \rightarrow i \rightarrow Q,\sigma \leftrightarrow -(x,i),label,cpens >\]
\[ \text{if}\ \{\text{rec a x} \rightarrow Q\} \\text{G} := \text{HF}(P) .\]

crl [b] :
\< P,\sigma,label,cpens > \rightarrow \{c\} \< P',\sigma,label <-[l,b],cpens >
\[ \text{if assign}(x,e) \rightarrow P' := \text{HF}(P) .\]

crl [c] :
\< P,\sigma,label,cpens > \rightarrow \{c\} \< P',\sigma,label <-[l,b],cpens >
\[ \text{if assignlabel}(1,b) \rightarrow P' := \text{HF}(P) .\]

crl [d] :
\< P,\sigma,label,cpens > \rightarrow \{c\} \< P',\sigma,label,cpens ^ C >
\[ \text{if compen}(C) \rightarrow P' := \text{HF}(P) .\]

crl [e] :
\< P,\sigma,label,cpens > \rightarrow \{c\} \< fault,\sigma,label,cpens >
\[ \text{if throw} := \text{HF}(P) .\]

crl [f1] :
\< P,\sigma,label,Y ^ X > \rightarrow \{c\} \< X;undo,\sigma,label,Y >
\[ \text{if undo} := \text{HF}(P) .\]

crl [f2] :
\< P,\sigma,label,cpens > \rightarrow \{c\} \< special,\sigma,label,empty >
\[ \text{if undo} := \text{HF}(P) .\]

5 Mechanizing the Generated Operational Semantics from Algebraic Semantics

In the last section, we studied the derivation of operational semantics from algebraic semantics. A derivation strategy was defined and a set of transition rules for each statement can be derived. From the derivation strategy, we can generate operational semantics in traditional style inductively based on the syntax of programs. We also implement them separately in Maude as transition rules. In this section we study the derived operational semantics implemented in Maude.

5.1 Sequential Composition

For sequential composition, the following four rules reflect its semantics. Special-\textit{aly}, when a program transits into the \textit{special} or \textit{fault} state, any program following it will never be executed. As illustrated in \textbf{Sequential-3} and \textbf{Sequential-4}, program \( P;Q \) will transit into \textit{special} or \textit{fault} in case that \( P \) does.

crl [Sequential-1] :
\< P;Q,\sigma,label,cpens > \rightarrow \{c\} \< Q,\sigma',label',cpens' >
\[ \text{if}\ \text{undo} := \text{HF}(P) .\]

crl [Sequential-2] :
5.2 Parallel Composition

In this section, we will explain how parallel composition works. First, when any one of the parallel components $P$ and $Q$ is nondeterministic, $P \parallel Q$ first perform a tau action as shown by Parallel-1-1 and Parallel-1-2.

**crl [Parallel-1-1]**

if $\text{< P;Q,sigma,label,cpens >} \Rightarrow \{\tau\} \text{< par(P',Q'),sigma,label,cpens >}$

if $\text{< P,sigma,label,cpens >} \Rightarrow \{\tau\} \text{< P',sigma',label',cpens' >}$

\$/\ P' \neq nil \$/\ P' \neq special \$/\ P' \neq fault .

**crl [Parallel-1-2]**

if $\text{< P;Q,sigma,label,cpens >} \Rightarrow \{\tau\} \text{< par(P',Q'),sigma,label,cpens >}$

if $\text{< P,sigma,label,cpens >} \Rightarrow \{\tau\} \text{< P',sigma',label',cpens' >}$

\$/\ Q, sigma, label, cpens > \Rightarrow \{\tau\} \text{< Q',sigma',label',cpens' >} .

In the normal case, when there is no communication between the parallel components, we apply the interleaving semantics to the parallel composition as shown by Parallel-2-1.

**crl [Parallel-2-1]**

if $\text{< P;Q,sigma,label,cpens >} \Rightarrow \{c\} \text{< par(P',Q'),sigma,label,cpens >}$

if $\text{< P,sigma,label,cpens >} \Rightarrow \{c\} \text{< P',sigma',label',cpens' >}$

\$/\ Q, sigma, label, cpens > \Rightarrow \{act\} \text{< Q',sigma',label',cpens' >}$

act $\neq$ tau /\ Q' $\neq$ fault .

The following rules define transitions when there are communications between processes. When one is to reply and the other is to receive through the same channel, they can do the communication and an assignment is used to represent the value passing. Otherwise, the communication will fail.

**crl [Parallel-3]**

if $\text{< P;Q,sigma,label,cpens >} \Rightarrow \{c\} \text{< x := y ; par(P',Q'),sigma,label,cpens >}$

if $\text{< P,sigma,label,cpens >} \Rightarrow \{x \# a \? m\} \text{< P',sigma',label',cpens' >}$

\$/\ Q, sigma, label, cpens > \Rightarrow \{y & a \! m\} \text{< Q',sigma',label',cpens' >} .

**crl [Parallel-4]**

if $\text{< P;Q,sigma,label,cpens >} \Rightarrow \{act\} \text{< fault,sigma',label',cpens' >}$

if $\text{< P,sigma,label,cpens >} \Rightarrow \{act\} \text{< fault,sigma',label',cpens' >}$
< Q, sigma, label, cpens > => \{ act' \} < Q', sigma'', label'', cpens'' > \land act' /=\tau.

5.3 Scope

In this subsection, we concentrate on the scope structure. For program \{ A ? C, F \}, if A has successfully terminated, then it will install the compensating program C into the compensation list by the action compen(C). On the other hand, if A has encountered a fatal error, the fault handler F will take in charge, as shown by Scope-2. Otherwise, A will continue to execute. Specially, if A is to do an undo, it will start the compensation mechanism until all the actions in the compensation list are executed, as shown in Scope-3.

crl [Scope-1-1] :
\[ < \{ A ? C, F \}, \sigma, \lambda, \kappa > => \{ \text{act} \} < \text{compen}(C), \sigma', \lambda', \kappa' > \]
< A, sigma, label, cpens > => \{ act \} < nil, sigma, label, cpens' > .

crl [Scope-1-2] :
< compen(C), sigma, label, cpens > => \{ act \} < nil, sigma, label, cpens ` C > .

crl [Scope-2] :
\[ < \{ A ? C, F \}, \sigma, \lambda, \kappa > => \{ \text{act} \}' < F', \sigma'', \lambda'', \kappa'' > \]
< A, sigma, label, cpens > => \{ act \} < fault, sigma, label, cpens ' > .

rl [Scope-2] :
\[ < \{ A ? C, F \}, \sigma, \lambda, \kappa > => \{ \text{act} \}' < F', \sigma'', \lambda'', \kappa'' > \]
< F, sigma, label, cpens > => \{ act \} < nil, sigma, label, cpens > .

As a basic element of compensation mechanism, we explain here the undo command.

rl [Undo-1]:< undo, sigma, label, empty > => \{ c \} < special, sigma, label, empty > .

rl [Undo-2]:< undo, sigma, label, Y ` X > => \{ c \} < X ; undo, sigma, label, Y > .

Until the compensation list is empty, the program will try to do the programs stored in the compensation list in reverse order whenever it encounters an undo action. In the end, it will transit to special state. These two transitions together with the scope structure compose the compensation mechanism for BPEL.

5.4 Linking between Derivation Strategy and Derived Operational Semantics

In the last section we considered the derivation of operational semantics from algebraic semantics. A derivation strategy was defined and the operational se-
mantics for each statement can be derived. Also in the last subsections we me-
chanized the derived operational semantics. This subsection studies the linking
between the derivation strategy and the derived operational semantics from the
mechanical viewpoint. Firstly we use an example to illustrate the equivalence
between the derivation strategy and the derived operational semantics. Second-
ly we discuss the mechanical approach for the proof of the equivalence.

**Example.** We use the first example described in section 3.6 to show the equi-
valance of the derivation strategy and the derived operational semantics. The
parallel program is shown below:

\[
\begin{align*}
&(\text{rec } a \ x ; \ \text{if } x <= 0 \ \text{then } x := -x \ \text{else } x := x + 1 ; \ \text{rep } a \ x) \| \\
&(\text{rep } a \ y ; \ \text{rec } a \ y)
\end{align*}
\]

We show one of the execution paths below. The initial values of \(x\) and \(y\) are set
to 0 and 5. empty represents the uninitialized state of Env, Labels, Copens in
the configuration.

\[
\begin{align*}
&\langle (\text{rec}'a'x ; \ \text{if } 'x<0 \ \text{then } 'x:=(-'x) \ \text{else } 'x:=('x+1) ; \ \text{rep}'a'x) \\
&\| (\text{rep}'a'y ; \ \text{rec}'a'y),'(x,0)|'(y,5),empty,empty > \rightarrow \langle 'x:=y ; \ \text{****** a first receive/reply sync} \\
&(\text{if } 'x<0 \ \text{then } 'x:=(-'x) \ \text{else } 'x:=('x+1) ; \ \text{rep}'a'x) \\
&\| (\text{rec}'a'y,('x,0)|'(y,5),empty,empty > \rightarrow \langle \text{assign}('x,'y) ; (\text{if } 'x<0 \ \text{then } 'x:=(-'x) \ \text{else } 'x:=('x+1) ; \ \text{rep}'a'x) \\
&\| (\text{rec}'a'y,('x,0)|'(y,5),empty,empty > \rightarrow \langle (\text{if } 'x<0 \ \text{then } 'x:=(-'x) \ \text{else } 'x:=('x+1) ; \ \text{rep}'a'x) \\
&\| (\text{rec}'a'y),('x,5)|'(y,5),empty,empty > \rightarrow \langle \text{rep}'a'x \| (\text{rec}'a'y => nil ),('x,6)|'(y,5),empty,empty > \\
&\| (\text{assign}('x,'y) ; \ \text{****** a second receive/reply sync} \rangle
\end{align*}
\]

There are two receive/reply synchronization in the path, which we have marked
in the above. At the end, the second program of the parallel process will be
blocked at the action rep. This path together with others are all searched out in
both the derivation strategy and derived operational semantics, and here we only
show this particular one shown above as an example, indicating the equivalence
of the derivation strategy and derived operational semantics for the example
parallel program.

**Discussion.** Now we study the mechanical proof of the equivalence of the deriva-
tion strategy and the derived operational semantics. Theoretically we have two
theorems to be proved.

(1) If transition aa exists in the transition system of the derived operational
semantics, then it also exists in the derivation strategy.

(2) If transition bb exists in the derivation strategy, then it also exists in the
transition system of the derived operational semantics.

The two theorems have been proved in our previous paper by theoretical analysis.
Since we have implemented the derivation strategy and the derived operational
semantics in Maude, we want an proof for the equivalence by machine. The derivation strategy and transition system are implemented in rewriting logic, we have several challenges to prove the above theorems in Maude ITP (i.e., an interactive theorem prover [15]). One of them is how to represent transitions in MEL. Our strategy is to declare two new sorts Pair and Ar with Ar as Pair’s subsort. And an operator is declared to represent transitions in the style of membership relations.

\[
\begin{align*}
\text{sort } & \text{Ar Pair} . \\
\text{subsort } & \text{Ar < Pair} . \\
\text{op } & \text{--> : Config Config -> Pair} .
\end{align*}
\]

The sort pair contains all the pairs of two configurations, but not all of them are correct transitions. So we can use the membership operator, declaring the correct transitions represented as pairs to be a member of sort Ar.

For theorem (1), we can declare all the transitions represented as Pairs in the derived strategy to be members of sort Ar.

\[
\begin{align*}
\text{fmod } & \text{DERIVATION-STRATEGY’ is} \\
\text{.....}
\end{align*}
\]

\[
\begin{align*}
& \text{cmb } [a1'] : < P,\sigma,\text{label},\text{cpens} > \rightarrow \{c\}< P,\sigma,\text{label},\text{cpens} > : \text{Ar} \\
& \quad \text{if } \{b \& \text{skip } \Rightarrow P\} \quad \text{GC := HF}(P,\text{null}) \land b[\sigma] . \\
& \text{cmb } [a2'] : < P,\sigma,\text{label},\text{cpens} > \rightarrow \{c\}< R,\sigma,\text{label},\text{cpens} > : \text{Ar} \\
& \quad \text{if } \{g(\text{label}) \Rightarrow R\} \quad \text{GC := HF}(P,\text{null}) \land g[\text{label}] .
\end{align*}
\]

Then all we have to do is to prove the transitions, also represented as Pairs, to be members of sort Ar.

\[
\begin{align*}
\text{(goal sequential1 : DERIVATION-STRATEGY’} \quad \text{is} \\
\text{.....}
\end{align*}
\]

\[
\begin{align*}
& \text{goal sequential1 : DERIVATION-STRATEGY’} \\
& \text{|- AP:Program ; Q:Program ; s:Status ; s’:Status} \\
& \text{((< P, s > \rightarrow \text{< nil, s’ >}: \text{Ar}) \Rightarrow ((< P ; Q, s > \rightarrow \text{< Q, s’ >}: \text{Ar})) .)
\end{align*}
\]

For theorem (2), the method is the same. Here, we give a strategy for proving the equivalence of derivation strategy and the derived operational semantics. Many implementation details for interactive proof in Maude ITP are a big challenge.

6 Related Work

Compensation is one typical feature for long-running transactions. Butler et al. have explored the compensation feature in the style of process algebra CSP [16], namely compensating CSP. The operational semantics and trace semantics have been studied [8, 7]. The compensation was introduced via a construct \( P \div Q \), where \( P \) is the forward process and \( Q \) is its associated compensation behavior. StAC (Structured Activity Compensation) [5] is another business process modeling language, where compensation acts as one of its main features. Its operational semantics has also been studied in [4]. Meanwhile, the combination of StAC and B method [1] has been explored in [6], which provides the precise description of business transactions. Bruni et al. have studied the transaction
π-calculus has been applied in describing web services models. Laneve and Zavattaro [21] explored the application of π-calculus in the formalization of the semantics of the transactional construct of BPEL. They also studied a standard pattern of Web Services composition using π-calculus. For verifying the properties of long-running transactions, Lanotte et al. have explored their approach in a timed framework [22]. A model of Communicating Hierarchical Timed Automata was developed where time was also taken into account. Model checking techniques have been applied in the verification of properties of long-running transactions.

Unifying Theories of Programming (abbreviated as UTP) was developed by Hoare and He in 1998 [19]. UTP covers wide areas of fundamental theories of programs in a formalized style and acts as a consistent basis for the principles of programming language. For relating operational and algebraic semantics, Hoare and He have studied the derivation of operational semantics from the algebraic semantics [18, 19]. An operational semantics of CSP [16] was derived, based on CSP’s algebraic laws according to a derivation strategy (called the action transition relation). An operational semantics of Dijkstra’s Guarded Command Language (abbreviated as GCL) was also derived based on GCL’s algebra according to the derivation strategy (called the step relation). The total correctness of the derived GCL’s operational semantics was also discussed in [20]. Recently, Hoare proposed the challenging research for the semantic linking between algebra, denotations, transitions and deductions [17].

7 Conclusion

Business process modeling languages are important for the web-based enterprise application. Our method is a new aspect of studying scope-based compensating and fault handling mechanism. In this paper, we studied the linking theory between algebraic semantics and operational semantics for BPEL. The linking was mechanized in rewriting logic system Maude.

First we implemented the syntax of BPEL in an obvious but accurate way thanks to the expressiveness of Maude. Then four typical forms for BPEL program is introduced, and the algebraic laws for sequential, parallel and scope structure are studied. The concept of summation was introduced. All the laws were encoded in Maude. Besides the algebraic laws, the head normal form of each program was defined and encoded in Maude. Based on the concept of head normal form, a strategy for deriving operational semantics from algebraic semantics was provided and a set of transition rules for each program can be generated. The generated operational semantics were implemented in Maude. Finally we used two examples to show the equivalence of the derivation strategy and the derived operational semantics.
For the future, we continue to explore the unifying theories for web services [19, 31], as well as the further web service models, including the probabilistic web service models [25, 13] and web service transaction models [24]. Further, we are also interested in how our mechanical approach can be applied to system verification.

References

Author Index

Arbab, Farhad, 80
Armas-Cervantes, Abel, 33
Barbosa, Luís S., 96
Demarty, Guillaume, 65
Dumas, Marlon, 33
García-Bañuelos, Luciano, 33
Hallé, Sylvain, 65
Le Breton, Gabriel, 65
Liu, Peng, 49, 112
Müller, Richard, 18
Maronnaud, Fabien, 65
Oliveira, Nuno, 96
Sürmeli, Jan, 3
Santini, Francesco, 80
Stahl, Christian, 18
Sun, Zailiang, 49
Toumani, Farouk, 1
Tuosto, Emilio, 2
Van der Aalst, Wil M. P., 18
Wu, Xi, 49
Zhang, Yue, 49
Zhao, Yongxin, 49
Zhu, Huibiao, 49, 112